Version 5 – Septembre 2023

Cours de Génie Logiciel

Vincent T'KINDT

Table des matières

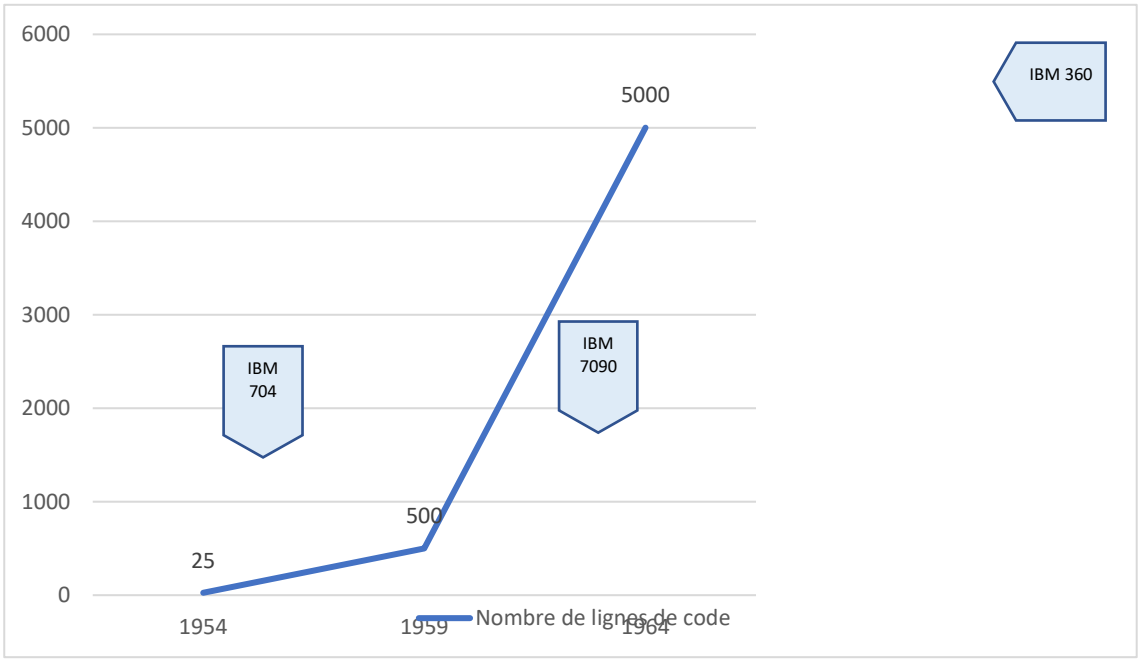
Chapitre 1 : Notion de Génie Logiciel

I – Généralités : échecs et enjeux

Le développement du Génie Logiciel est étroitement lié à l'évolution de l'informatique. Revenons dans les années 1950-1960 et regardons l'évolution des ordinateurs à cette époque (Figure 1). Les progrès fait sur un plan matériel sont fulgurants avec des ordinateurs capables d'accueillir des logiciels de plus en plus volumineux.
Dans les années 1950, on pouvait faire un logiciel en faisant une analyse rapide du problème puis en consacrant la majeure partie du temps à coder. Il s'agit d'une approche très intuitive du développement.
Vers la fin des années 1960, un triste constat est fait : certes on fait des logiciels de plus en plus volumineux, mais avec un certain nombre de problèmes. Citons :

- Les estimations de coût et délai se révèlent inférieures à la réalité (d'un facteur 2 à 4),
- Les équipes de programmeurs sont confrontées à des problèmes de communication que l'encadrement peine à résoudre,
- Les logiciels produits en répondent pas toujours aux attentes.
- Quand un logiciel est livré, un gros effort est nécessaire pour corriger ses bugs (plutôt que de le faire évoluer).

Le matériel avait évolué mais, paradoxalement, pas les méthodes de développement ! La fin des années 1960 a donc été la période de la *crise du logiciel.*



- Figure 1 : Evolution des capacités des ordinateurs en termes de taille des logiciels -

Pour élaborer des systèmes logiciels de qualité et rentables à moyen et long terme il faut établir des *règles de développement* qui permettent de répondre aux préoccupations suivantes:

- Comment réduire le coût d'un logiciel ?
- Comment augmenter la qualité des logiciels ?
- Comment satisfaire au mieux les exigences client (délais, prix, ...) ?

Malheureusement il n'existe pas une méthode de développement universelle. Chaque développement procure des connaissances sur le domaine d'application du logiciel (exemple : industrie pétrochimique, comptabilité, ...). La capitalisation de cette expérience nous conduit à particulariser nos méthodes à ces domaines. Dans ce cours nous ne pouvons donc que présenter des traits communs, donc généraux, aux développements de logiciels.

II – Définition et objectifs du Génie Logiciel

La conception d'un logiciel est séparée en deux grandes activités distinctes : la gestion de projets et le génie logiciel [cours CP][Menthonnex-1996a]. Le Génie Logiciel a fait l'objet de premiers travaux fondateurs dans la période 1968-1975 et est encore aujourd'hui en pleine évolution.

La définition du Génie Logiciel que nous retiendrons est donnée dans [Jaulent-1992] :
« *Le Génie Logiciel est l'ensemble des procédures, méthodes, langages, ateliers, imposés ou préconisés par les normes adaptées à l'environnement d'utilisation afin de favoriser la production et la maintenance de composants logiciels de qualité* ».

Du point de vue de la loi (arrêté ministériel du 22/12/1981) un logiciel est : « *l'ensemble des programmes, procédés et règles, et éventuellement de la documentation, relatifs au fonctionnement d'un ensemble de traitements de l'information* ».
Sur le plan législatif un logiciel ne se résume pas seulement au code produit mais est constitué également de la documentation et des procédés utilisés pour l'élaboration du code (retirer la mention *éventuellement*).

Définition officielle du Génie Logiciel : (arrêté ministériel du 30/12/1983)
« *Le Génie Logiciel est l'ensemble des activités de conception et de mise en oeuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi* »

Reprenons certains points de la définition du Génie Logiciel :

- ⇨ **Procédure = cycle de vie** : C'est l'ensemble des étapes à suivre pour créer un logiciel. Exemple : cycle en V, cycle incrémental, …
- ⇨ Dans un cycle de vie, on utilise des **méthodes** (ou méthodologie pour représenter un problème, des cas d'utilisation d'un logiciel, une architecture logiciel). Exemple : UML, Merise, …
- ⇨ Le logiciel est programmé dans un **langage** à partir de sa conception papier. Le choix d'un langage n'est pas trivial car il influe sur les méthodes à utiliser et rend plus simple ou plus complexe l'écriture du programme.

ILLUSTRATION : Une étude menée dans les années 1980 (voir [Knight et al - 1986]) montre que lors du codage d'un logiciel le nombre moyen de « bugs constatés » pour 100 lignes de codes est 4 fois moins important an ADA qu'en Fortran..

Développer en suivant une démarche de Génie Logiciel vise à satisfaire plusieurs objectifs :

- Maîtriser les coûts et les délais,
- Produire des logiciels de qualité plus facilement maintenables.

Suivre une démarche de Génie Logiciel ne vous fera pas gagner du temps/argent sur du court terme (par rapport à une approche non structurée) mais à moyen et long terme le gain est énorme.

III - Les ateliers de GL

Un Atelier de Génie Logiciel (AGL, ou *CASE* en anglais pour « *Computer Aided Software Engineering* ») est une suite logicielle permettant la production du logiciel tout au long de son cycle de vie. Plus il est complet et plus il va couvrir le cycle de vie du logiciel.

Un AGL poursuit plusieurs objectifs :

- Augmenter la productivité d'une équipe de développement,
- Améliorer la qualité des logiciels produits,
- Aider l'équipe à appliquer les différentes normes, procédures, incontournables étapes dans le processus de développement,
- Soulager l'équipe des tâches fastidieuses et répétitives telles que les vérifications de cohérence lors des phases de spécification et conception du logiciel.

NOTE : Il faut se méfier du terme « Atelier de Génie Logiciel » qu'on a tendance à utiliser pour qualifier des outils qui n'en sont pas réellement. Un compilateur munit d'un debugueur ne constitue pas un atelier de Génie Logiciel.

Quelques exemples d'AGL : ArgoUML[1], IntelliJ IDEA[2], PowerDesigner de SAP[3], Rational Rose d'IBM[4].

IV –Le Génie Logiciel et la Conduite de Projets

La Gestion de Projets ou Conduite de Projets vise à rationnaliser l'activité de conception d'un projet logiciel. Cela implique plusieurs sous-activités, dont notamment :

1. **Planification**. On doit être capable de découper le projet en tâches/activités/phases qui sont planifiées dans le temps. Cette planification est mise à jour au fur et à mesure que le projet avance.
2. **Budgétisation**. Cela inclus, l'étude préalable des coûts, des sources de financement, du retour sur investissement, la prise en compte des aléas par rapport au planning, …
3. **Gestion du risque**. Il faut être capable de gérer les aléas et savoir anticiper les risques notamment en anticipant la formation des membres du projet, en évaluant l'utilité de la sous-traitance (off-shore), …
4. **Gestion de la qualité**. Il faut être capable de définir un référentiel qualité et de vérifier en cours de projet que l'on suit bien ce référentiel.
5. **Structuration de l'équipe de développement**. Il faut être capable en amont du projet de créer et structurer correctement l'équipe de développement, et il faut être capable en cours de projet d'adapter l'équipe à l'évolution du projet.

[1] argouml-tigris-org.github.io | ArgoUML resources and web pages.
[2] IntelliJ IDEA – the Leading Java and Kotlin IDE (jetbrains.com)
[3] SAP PowerDesigner | SAP Help Portal
[4] www.ibm.com

Typiquement, une équipe de développement peut être constituée des membres suivants (en totalité ou en partie), selon la terminologie proposée par le Syntec :

- ***Chef de projet***. Il conduit le projet de bout en bout et en assume la responsabilité des différentes phases. Il constitue l'équipe, coordonne le travail des différents intervenants, ajuste en permanence le planning en fonction de l'avancement. Son objectif est de terminer le projet en fournissant des livrables de qualité, conformes aux cahiers des charges définis avec le client, en respectant délais et budget.
- ***Assistant Chef de projet.*** L'assistant chef de projet est chargé de seconder le chef de projet dans les tâches liées à la conduite d'un projet, il est le relais opérationnel du chef de projet.
- ***Analyste Fonctionnel***. La mission d'un analyste fonctionnel consiste à concevoir fonctionnellement une application (ce qu'elle fait). Pour cela, il doit analyser les besoins des utilisateurs et les formaliser avant de proposer les solutions les mieux adaptées.
- ***Développeur***. En fonction de l'analyse des besoins des utilisateurs et de l'étude fonctionnelle, le développeur va participer à la réalisation de la phase de *Conception du Logiciel* et programmer dans un langage particulier l'application attendue, tout en respectant délais et normes de qualité définis.
- ***Ingénieur d'intégration***. Il rassemble les différents composants du développement sur une plate-forme d'intégration. Cette tâche repose sur une importante phase de tests respectant une méthodologie et des procédures rigoureuses : spécification, planification, réalisation, bilan. Il s'agit notamment de vérifier la compatibilité entre les différents éléments, logiciels, matériels ou systèmes. Une fois les tests effectués, l'ingénieur intégration remédie ou fait remédier aux bugs et aux erreurs.
- ***Ingénieur qualité méthode ou Responsable Qualité***. Il définit et met en place les procédures qualité et les méthodes à utiliser. Il s'assure qu'elles sont respectées. Son expertise sur le sujet lui permet d'informer et de convaincre de l'intérêt et de l'utilité ce ces choix en matière de qualité et de méthodes.
- ***Architecte technique.*** Il a en charge la définition de l'architecture technique du système d'information, en veillant à la cohésion entre les aspects matériels, applicatifs, systèmes d'exploitation, réseaux…
- ***Consultant en technologie***. Ils sont amenés à faire profiter de leur expertise pointue des entreprises clientes souhaitant mener à bien des projets complexes. Leur mission débute généralement dès la phase de spécifications pour se terminer par les tests, la validation et le suivi du déploiement.

Le Génie Logiciel concerne les activités liées à l'aspect technique du développement : Comment spécifier l'architecture du logiciel ? Comment réaliser des tests ? Comment coder proprement ? Les membres de l'équipe de développement qui vont être amenés à appliquer du Génie Logiciel sont plus particulièrement l'***Analyste Fonctionnel***, le ***Développeur***, l'***Ingénieur d'Intégration***, et le ***Consultant en technologie***.

 **Vous travaillerez tout ça dans le cours de Gestion de Projets au S7 (DI4).**

V – La méthodologie UML

UML veut dire *Unified Modeling Langage* et est issu de la fusion et l'unification de trois de

ces méthodes (1994-1996) : OOD (G. Booch), OMT (J. Rumbaugh), et OOSE (I. Jacobson).

La version 1.0 a été adoptée, en 1997, par l'OMG[5] (Object Management Group) qui est une association à but non lucratif pour la standardisation/promotion de l'approche Orientée Objets. Parmi ses membres figurent EADS, NASA, HP, Oracle, Microsoft, IBM… L'OMG conçoit et rédige des spécifications (UML et autre). Aujourd'hui, UML est la norme en termes de méthodologie orientée objets. La dernière version est la 2.5.1 et date de décembre 2017.

Certes, UML a été introduite pour être utilisée lorsqu'on développe un logiciel en suivant une approche orientée objets : néanmoins, nombreux de ses diagrammes ne sont pas propres à l'orientée objets et peuvent être utilisés même si au final le logiciel sera développé dans un langage impératif comme le C, par exemple.

Pour cette raison, dans le cours de Génie Logiciel nous verrons un certain nombre de diagrammes UML applicables sans pour autant être dans une approche orientée objets.

A quoi va servir UML ?

- Comprendre et décrire formellement les besoins
- Les faire valider !
- Spécifier des systèmes
- Concevoir et construire des solutions
- Documenter un système,
- Communiquer entre les membres de l'équipe projet en utilisant un langage commun

Le résultat de l'utilisation de UML est **un modèle**, constitué de **plusieurs diagrammes** constituant chacun une vue différente du système. Dans la version 2.x d'UML, on dénombre 14 diagrammes différents : mais tous n'ont pas forcément vocation à être utilisés pour votre projet.

Sous UML on distingue deux familles de diagrammes : les diagrammes structurels (statiques) et les diagrammes comportementaux (dynamiques).

Diagrammes structurels :

- **Diagramme de classes**
- Diagramme d'objets
- Diagramme de composants
- Diagramme de déploiement
- Diagramme de profils
- *Diagramme de paquetages (package)*
- *Diagramme de structures composites*

Diagrammes comportementaux :

- **Diagramme de cas d'utilisation**
- **Diagramme d'activités**
- Diagramme d'états-transitions
- **Diagramme de séquence**
- Diagramme de communication
- *Diagramme d'interaction d'ensemble*

[5] OMG | Object Management Group

- *Diagramme de temps*

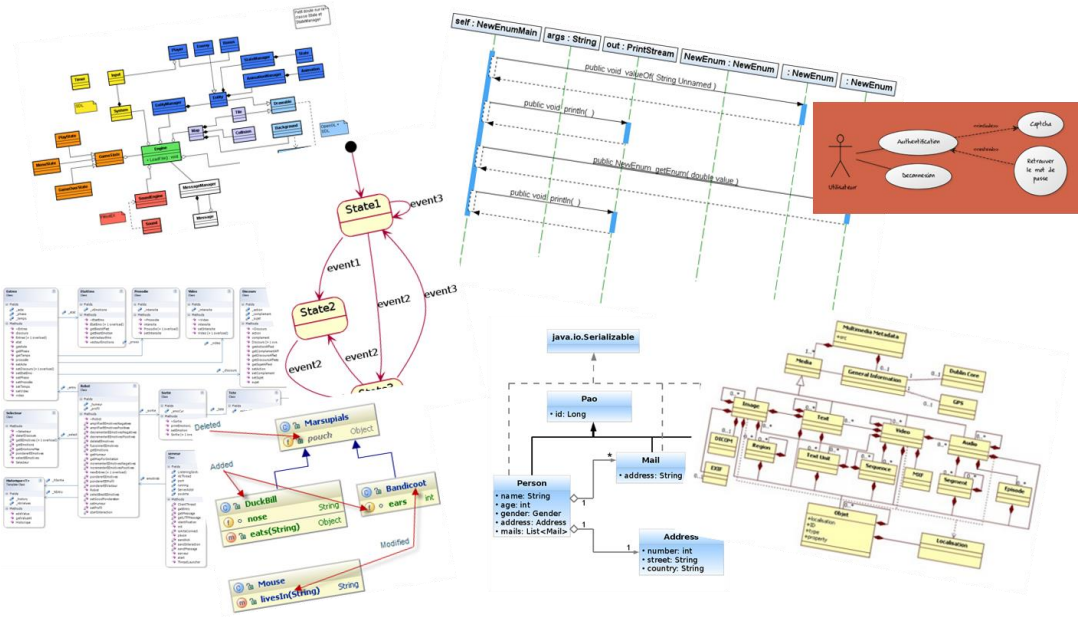
Quelques règles d'utilisation des diagrammes UML :

- **Tout est optionnel !**
  - Certains diagrammes seront inapplicables à des domaine spécifiques,
  - Certains diagrammes n'apportent parfois aucune information pertinente,
  - Même pour un seul diagramme, la notation est très dense et variée : ne vous sentez pas obligés de tout utiliser,
  - Il faut surtout éviter de charger inutilement les diagrammes : à trop vouloir les détailler vous pourriez les rendre moins compréhensibles…
  -

Un bon diagramme doit apporter une information claire, exploitable et compréhensible par l'ensemble des interlocuteurs.

- Il est courant d'avoir plusieurs diagrammes représentant la même information mais sous des angles différents ou de façon plus ou moins détaillée

- Vos modèles seront rarement complets
  - Ils seront revus, corrigés, complétés au fur et à mesure du projet
  - L'essentiel est qu'ils ne contiennent que les détails nécessaire *le jour J*



Le diagramme de classes sera introduit dans l'UE Conception et Programmation Orientée Objets 1 au S6 (DI3).

Chapitre 2 :Le cycle de vie du logiciel

I – Le cycle de vie d'un logiciel

Le *cycle de vie* d'un logiciel définit l'ensemble des phases que l'on doit suivre pour créer le logiciel. On distingue généralement le *cycle de vie,* du *cycle de développement* qui est plus large : ce dernier étend le cycle de vie pour y introduire des phases typiques à la gestion de projets (planification, budgetisation, gestion des risques, définition d'un référentiel qualité, …).

On peut trouver de nombreux cycles de vie : le cycle en cascade, le cycle en V, le cycle incrémental, … Au fond, ils reviennent tous au même, c'est-à-dire qu'ils mettent tous en jeux les mêmes activités intellectuelles. Ce qui diffère est « *quand on fait quoi* ». En fait, lorsqu'on crée un logiciel on doit toujours réaliser les activités suivantes :

- Définition ou Spécification (reformulation du cahier des charges pour établir les spécifications du système en tant qu'entité),
- Conception (passage des spécifications du système à une définition au plus bas de la réalisation),
- Réalisation (développement du logiciel et intégration dans l'infrastructure matérielle),
- Production (évaluation du système obtenu pour validation et mise en production),
- Fonctionnement (développement en série, commercialisation, maintenance).

II – Le Cycle en V

1°) Définition d'un système

Même si le Génie Logiciel ne s'intéresse a priori qu'au développement de *logiciels* , l'entité manipulée au début du processus est appelée *système* car elle est constituée d'une partie matérielle et d'une partie logicielle. Selon le SWEBOCK[6], un système est « *un ensemble d'éléments interagissant entre eux pour la réalisation d'un objectif bien défini. Cela inclus le matériel, le logiciel, le firmware, les intervenants, les données, les techniques, les services utilisés et tout autre moyen ayant contribué à la réalisation de cet objectif.* ».

NOTE : L'étude du système ne sera présentée que dans le cadre des deux premières phases du cycle en V. Ensuite les parties matérielles et logicielles font l'objet de développements séparés. C'est à la fin du cycle en V que les deux parties sont réunies. Pour le développement d'un système complet, se reporter à la méthodologie MCSE [Calvez-1990]

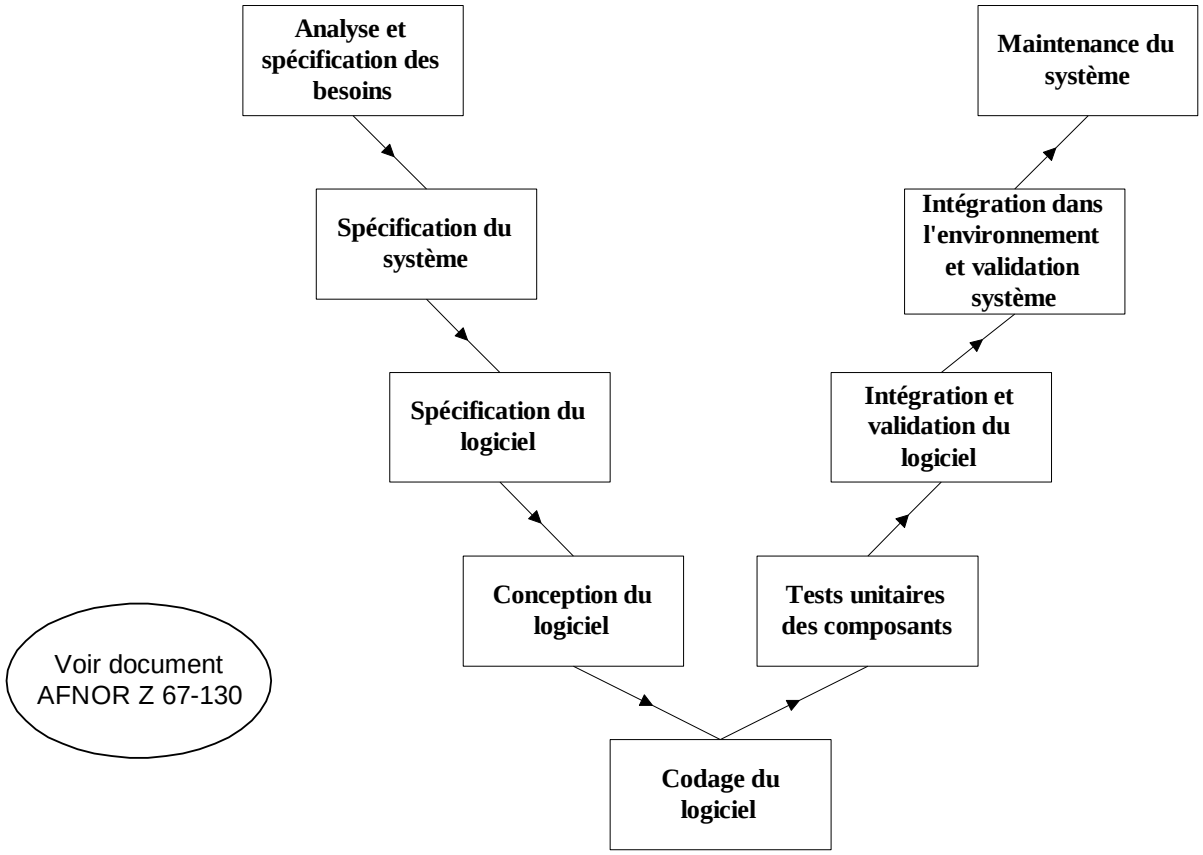
Un système prend place dans un environnement avec lequel il est en interactions permanentes. Cet élément est fondamental car il implique que le développement doit être orienté *utilisateurs d'abord*. Nous reviendrons sur cet aspect plus tard.

[6] Guide to the Software Engineering Body of Knowledge (SWEBOCK), IEEE Computer Society, Software Engineering Course (SWEBOK)| IEEE Computer Society.

2°) Présentation du cycle en V

Ce modèle possède l'avantage d'être normalisé et très pédagogique. Il est également très connu et utilisé.

NOTE : Il existe différentes versions du modèle en V avec un vocabulaire et un découpage en phases différent. Le modèle présenté dans la figue 3 est plus ou moins standard.



-Fig. 3 : Cycle en V-

L'explication des phases est la suivante :

Phase	Nom	Principales tâches	Documents produits	Qui écrit le document ?
1	Analyse et Spécification des besoins	Le Client écrit ses besoins dans le Cahier des Charges	Le Cahier des Charges	Le Client
2	Spécification du système	L'Equipe reformule le Cahier des Charges et fait affiner au Client son besoin.	Le Cahier de Spécification du Système	L'Equipe
3	Spécification du logiciel	L'Equipe doit modéliser l'architecture globale du logiciel et son comportement (pex diagrammes de classes, …)	Le Cahier de Spécification du logiciel	L'Equipe

4	Conception du logiciel	L'Equipe doit détailler l'architecture du logiciel, écrire les algorithmes des fonctions, définir les types de données, …	Le Cahier de Conception du Logiciel	L'Equipe
5	Codage	L'Equipe doit écrire dans un langage de programmation du logiciel à partir de la conception préalable	Le Cahier de Codage	L'Equipe
6	Tests unitaires	L'Equipe doit tester individuellement chaque fonction du programme pour vérifier leur bonne exécution.	Le Cahier des Tests Unitaires	L'Equipe
7	Intégration et validation du logiciel	L'Equipe doit tester le logiciel dans ses grandes fonctionnalités pour vérifier que les traitements réalisés sont cohérents.	Le Cahier d'Intégration et Validation du Logiciel	L'Equipe
8	Intégration et validation du système	L'Equipe doit tester le logiciel sur son architecture matérielle chez le Client. Une pré-version du logiciel peut être laissée en test chez le Client.	Le Cahier d'Intégration et Validation du système	L'Equipe
9	Maintenance	Le logiciel est opérationnel et entre dans sa phase de maintenance.	----------	----------

Si l'on regarde plus dans le détail le cycle en V on constate qu'il est découpé en deux parties :

1) ***une partie descendante*** dans laquelle le système est analysé et décomposé. On passe du besoin du Demandeur (phase d'Analyse et Spécification des besoins) à des composants logiciels unitaires (phase de codage),
2) ***une partie ascendante*** qui consiste à recomposer le système. On agrège ainsi les différents composants logiciels (phase de codage) pour construire un système opérationnel (phase de maintenance).

Chaque phase de la partie descendante est liée à la phase en vis-à-vis dans la partie de la phase ascendante. Ces liaisons servent au processus de *Certification* : on certifie qu'une phase descendante est correcte par comparaison avec la phase ascendante correspondante.

EXEMPLE : La phase des tests unitaires des composants est liée à celle de conception du logiciel. On dit alors que la conception du logiciel est certifiée lorsque :

a) les résultats des tests unitaires auront été comparés aux définitions des fonctions, procédures et modules de la phase de conception du logiciel,

b) cette comparaison fait ressortir que ces fonctions et procédures font réellement ce qui a été prévu qu'elles fassent.

Par ailleurs, le cycle en V met en jeu un autre processus pour garantir la fiabilité du produit obtenu : la *Validation*. Comme nous l'avons déjà vu, une phase du cycle en V est considérée comme une boîte noire dont les objectifs sont posés dans les documents (cahiers, prototypes,...) d'entrée. De même les documents de sortie d'une phase rendent compte de ce qui a été fait dans la phase et doivent répondre aux objectifs d'entrée. Le processus de

validation consiste à s'assurer, par comparaisons des documents d'entrée et de sortie, que le travail de la phase en cours a été fait entièrement et correctement.
Les processus de Validation et Certification sont capitaux dans le cycle en V car ils garantissent l'adéquation du système final aux besoins du Demandeur. Néanmoins, ils s'avèrent très contraignants. Le risque direct de la non-application de ces deux processus réside dans la présence de « bugs » qui ne seront détectés que tard dans le cycle en V. Les coûts additionnels seront alors élevés.

III - Remarques sur le processus de développement

1°) Utilisation du prototypage

On distingue deux types de prototypage : le *prototypage jetable* et le *prototypage évolutif*. Un prototypage jetable est un « squelette » du logiciel final qui n'est créée que dans un but et dans une phase particuliers du développement. A l'inverse le prototypage évolutif est conservé tout au long du cycle de vie. Il est amélioré et complété pour obtenir le logiciel final. Le modèle en spirale utilise ces deux aspects du prototypage. Tôt dans le développement l'utilisation du prototypage jetable permet de faciliter la définition des besoins du Demandeur puisque la communication Demandeur-Fournisseur est simplifiée. A partir du moment où le travail à réaliser a été clairement défini et compris, il peut être préférable d'utiliser le prototypage évolutif pour construire le logiciel demandé. Cela permet de faciliter le processus de développement [Bass et al.-1995] :

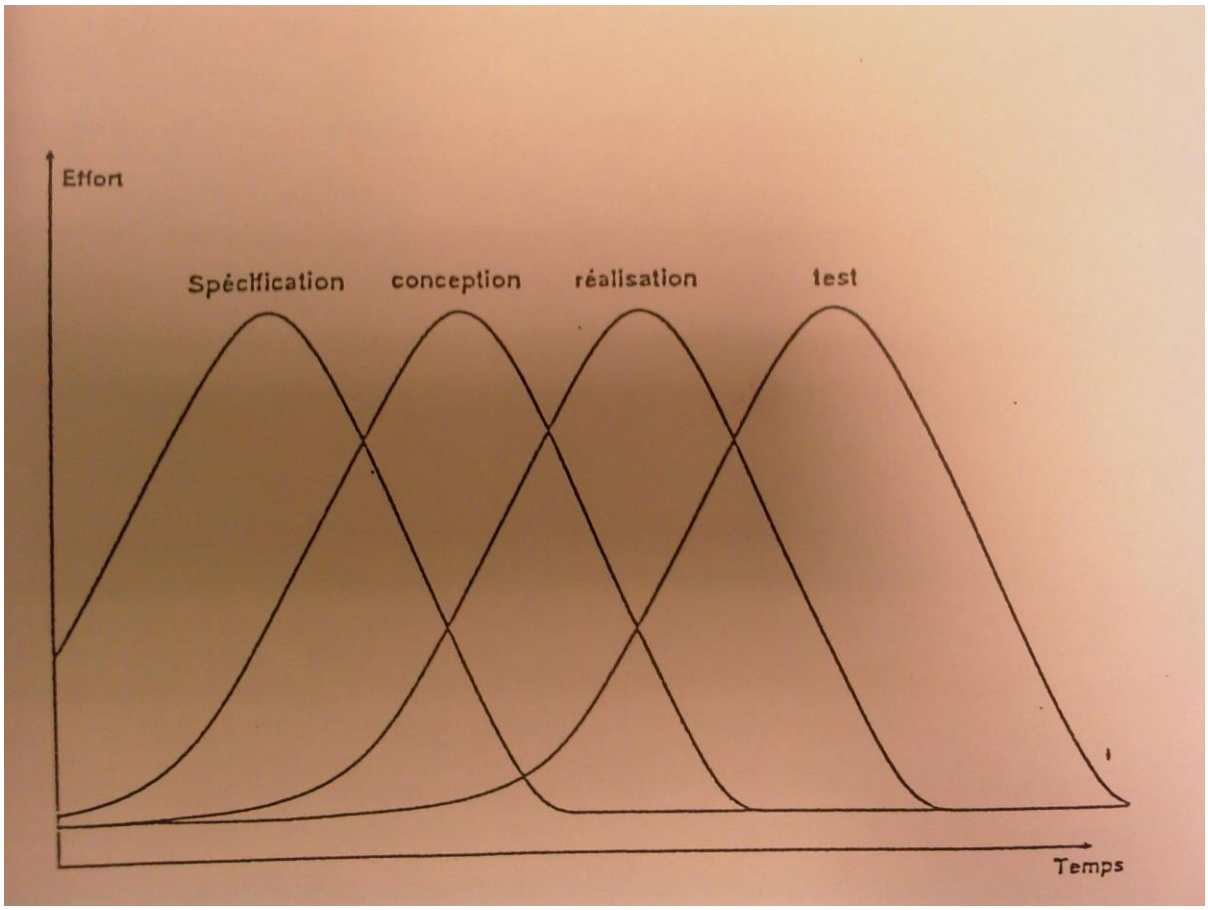
1) en faisant apparaître les problèmes potentiels liés aux performances du logiciel bien plus tôt,
2) le système est exécutable plus tôt dans le cycle de vie et la fidélité du logiciel par rapport aux spécifications augmente au fur et à mesure de l'amélioration du prototypage,
...

Une utilisation intéressante du prototypage consiste à hybrider une approche de type modèle en V et le prototypage jetable. [Gaudel et al-1996]

2°) L'enchaînement des phases

Le processus de développement n'est pas aussi séquentiel que dans les modèles présentés. A chaque fois que l'on crée un logiciel, on mélange implicitement toutes les phases à des degrés différents d'implication (voir figure 5). Ainsi la notion de phase correspond plus à un investissement maximum, à un instant donné, dans une activité particulière.

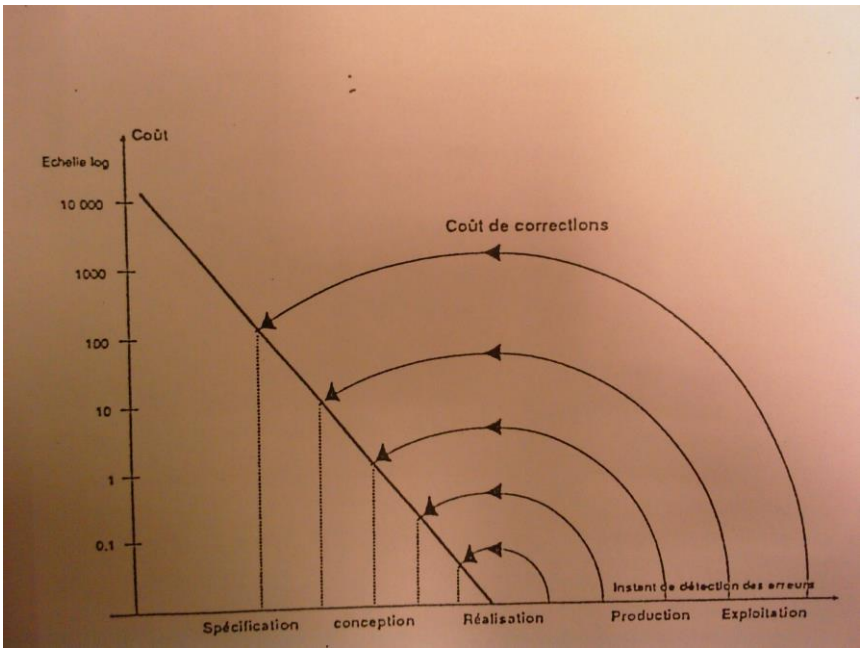
EXEMPLE : En phase de Spécification Système on peut déjà prévoir des campagnes de tests qu'il faudra impérativement réaliser. Les phases correspondent, en réalité, aux pics de la figure 5. Elles peuvent donc toutes commencer, plus ou moins, au même moment mais avec un investissement différent.



-Figure 5 : Recouvrement possible des phases -

3°) Coût des erreurs

Dans tout cycle de vie, les retours en arrière dans le développement sont autorisés. Néanmoins, ils ont des conséquences en termes de coût qui peuvent être désastreuses. La figure 6 (voir [Calvez-1990]) présente un diagramme de correspondance entre coûts additionnels et phases auxquelles des erreurs sont détectées. Ainsi, si une erreur survient en phase d'exploitation alors la cause peut être trouvée dans les spécifications et nécessite un retour à cette phase. Ce diagramme suppose qu'une erreur à une phase X n'est pas détectable dans les phases précédentes.
Toute activité de conception étant sujette à des erreurs il faut s'astreindre impérativement à vérifier son travail et le faire vérifier par d'autres membres de l'équipe de travail.



- Figure 6 : Courbe du coût de correction des erreurs -

4°) Répartition de l'effort

Concevoir un logiciel ce n'est pas courir un 100 mètres. Bien souvent l'approche intuitive consiste à mettre l'accent sur la phase de codage qui n'est que l'aboutissement des phases précédentes d'analyse. Un certain nombre d'ingénieurs et techniciens qui conçoivent des systèmes passent rapidement sur les phases d'analyses pour passer au codage. Cela est non seulement une source d'erreurs mais aussi d'incompréhensions du problème posé. Et donc une source de surcoûts en fin de développement.

ILLUSTRATION : [Calvez-1990] préconise de passer entre 20 et 30 % de son temps de travail dans les phases de spécification, uniquement pour modéliser le système et la solution que l'on propose en terme fonctionnel.

Chapitre 3 : Analyse et Spécification des Besoins

I – Définir pour le Client ses besoins

La phase d'analyse et spécification des besoins est la période pendant laquelle le Client et le Fournisseur prennent contact. Durant cette phase le Client (une personne, une société,...) exprime, dans ses termes, son besoin au travers du cahier des charges. Le Fournisseur doit alors s'exprimer sur ce cahier des charges, se prononcer sur sa faisabilité, demander des précisions... pour en fin de compte accepter le projet ou le refuser.

Le cahier des charges est rédigé dans les termes du client (le Demandeur) qui ne possède pas forcément des compétences en informatique. A l'inverse le Fournisseur (l'équipe de travail) ne possède pas forcément le vocabulaire et les connaissances liées au domaine d'activité du Client. Cette barrière de langage, de connaissances, est un des points les plus délicats de cette phase.

Le Cahier des Charges contient l'expression du besoin du Demandeur. Il devrait être rédigé par ce dernier et doit nécessairement décrire l'environnement dans lequel doit se trouver le système, les objectifs que celui-ci doit satisfaire ainsi que ses propriétés et contraintes. Il a pour objectif de répondre à la première question essentielle : Pourquoi un tel système ? Quel est le besoin ?

II – Rédaction du Cahier des Charges

Le contenu du Cahier des Charges est très variable d'un Client à l'autre (très succinct, très volumineux, ...). Comme guide à la rédaction d'un tel document on peut utiliser la norme AFNOR x 50.151.

Plan du Cahier des Charges

1 - Présentation du document
- Présentation du Client,
- Organisation du document,
- Conventions et terminologie utilisés dans le document.

2 - Présentation du produit
- Présentation générale du problème,
- Domaine d'application, marché et objectifs,
- Quels seront les utilisateurs du logiciel et leur lien,
- Quel est le besoin logiciel,
- Quelles seront les conditions de mise en fonctionnement du logiciel,
- Quelle est la maintenance attendue.

3 - Description de l'environnement
- Quelles sont les entités qui inter-agissent avec le logiciel (utilisateurs, autres logiciels,…) : caractéristiques, droits d'accès,…,
- Informations liées aux entités: Quelles sont les relations entre eux ?,
- Quelles sont les contraintes autres ?

4 - Description des fonctions que le logiciel doit satisfaire,

- Les fonctions principales à développer,
- Les fonctions complémentaires qui peuvent être à développer maintenant ou dans le futur,
- Configurations, options, variantes,
- Caractéristiques et performances attendues pour le logiciel.

5 - Contraintes

- Définition des contraintes d'interfaces (ergonomie du logiciel),
- Définition des contraintes de temps,
- Définition des contraintes d'utilisation,
- Définition des contraintes de réalisation (langage imposé, sous-traitance, …),
- Maintenance et évolutivité du logiciel attendus,
- Fiabilité et sûreté de fonctionnement liées à l'exécution du logiciel.

6 - Documentation

- Documents de spécification, de conception, ... attendus,
- Manuels utilisateurs, procédures d'installation, gestion du système…attendus.

7 - Plan de certification

- Critères d'appréciation du logiciel produit,
- Situations de tests et scénario à réaliser par l'Equipe,
- Limites d'acceptation, flexibilité.

8 - Plan de développement

- Proposition de planning du développement,
- Support pour le développement.

Note : Le *Cahier des Charges* et le *Cahier de Spécification du Système* sont des documents très similaires en termes de contenu (l'un est rédigé par le Client et l'autre par l'Equipe). Cela veut dire que pour rédiger le *Cahier des Charges* on peut reprendre le plan du *Cahier de Spécification du Système*.

Chapitre 4 : Analyse et Spécification du Système

I - Objectifs

Cette seconde phase débute dès qu'un accord contractuel (s'il y a lieu) entre le Demandeur et l'équipe de travail a été signé. Le Cahier des Charges n'exprime que le besoin brut du Client et est parfois incomplet : on parle dans ce cas des *besoins cachés* du Client qui n'ont pas été encore exprimés.
A ce stade du projet, l'erreur en termes de coûts et délais peut s'avérer importante et on ne maîtrise pas forcément les conditions dans lesquelles va se dérouler le développement. Il est donc nécessaire de reformuler, et détailler, dans des termes propres à celui-ci le Cahier des Charges pour aboutir au Cahier de Spécification du système. Ce travail doit être effectué conjointement par le Demandeur et l'équipe de travail au cours de réunions pilotes et d'une réflexion menée par l'équipe. Il doit permettre de mettre à plat tout le système et de lever d'éventuelles ambiguïtés. C'est le travail de la phase de spécification système.

A ce stade du développement, différents problèmes peuvent apparaître :

- Interprétations différentes du Cahier des Charges entre Demandeur et équipe de travail,
- Les souhaits du client peuvent évoluer au cours du développement,
- L'idée qu'a l'équipe de travail du problème peut évoluer dans une direction totalement différente de ce que désire le Demandeur,
- Sans retour de l'équipe de travail, le Demandeur peut progressivement être amené à douter du résultat du développement.

Cette phase est certainement la phase la plus « dangereuse » du cycle de vie. En effet, mal comprendre les besoins c'est arriver en fin de développement à un système inadéquat pour le client. Selon **Ramamoorthy** [Ramamoorthy et al-1987], 30% des erreurs apparaissant dans les phases de tests et de mise en fonctionnement sont dues à une mauvaise compréhension du problème et très souvent à la non-complétude d'expression des besoins dans le Cahier des Charges.

Une bonne spécification du système (et il en va de même pour les spécifications du logiciel) doit répondre aux trois exigences suivantes :

- Lisibilité : claire, sans ambiguïtés, écrit dans un langage accessible aux deux parties,
- Testabilité : les Concepteurs et le Demandeur doivent pouvoir vérifier la conformité de la réalisation aux spécifications,
- Maintenabilité : comme les besoins ont généralement tendance à évoluer durant le développement, les spécifications doivent pouvoir être aisément modifiées.

Les acteurs de cette phase sont diverses : on distingue entre la **Maîtrise d'Oeuvre** et la **Maîtrise d'Ouvrage**. La MOE regroupe les personnes qui ont en charge le développement informatique (Chef de Projet, Consultants, Architectes, ..) tandis que la MOA regroupe les acteurs du demandeur (Chef de Projet métier, Consultants métiers). Par ailleurs au sein de la MOE on trouve des Consultants métiers à mi-chemin entre l'informatique et l'expertise

métier : ils vont le lien entre le discours du Demandeur (exprimé dans des termes non informaticien) et le discours des informaticiens.

II – La démarche : comment comprendre les besoins du Client ?

La démarche est centrée sur les utilisateurs ! C'est le point clef. Vous allez vous poser plein de questions et dont un certain nombre auront leur réponse dans le Cahier des Charges. Pour les autres il faudra interroger votre Client : mais en tout cas, ne prenez jamais une décision à sa place !

Il faut donc tout d'abord se poser la question de **QUI** ou **QUOI** va interagir avec le **système** à concevoir par la suite. L'objectif est **d'identifier les Acteurs** (utilisateurs humains, logiciels existants, …) qui vont interagir avec le système. On va donc créer la *délimitation environnement-système*.

Ensuite, il faudra se poser la question de savoir ce que chaque Acteur va faire avec le système : quelles **fonctionnalités** réalisées ? Quels **objets** échangés ? Pour dresser le panorama de ces fonctionnalités et objets on peut très bien faire *des diagrammes de cas d'utilisation et des diagrammes d'activité UML*.

ILLUSTRATION : Votre Client est Polytech Tours et vous sollicite pour développer un logiciel qui permette de gérer les emplois du temps d'une école d'ingénieurs : on a donc une spécialité (informatique, ….), des cours (CMs, TDs, TPs), des étudiants répartis par groupes éventuellement, des salles et des professeurs. L'objectif est que le système permette aux scolarités de créer les emplois du temps de leur spécialité et aux étudiants de consulter leur emploi du temps. Evidemment, ce logiciel va s'appuyer sur le système d'information existant au sein de l'école, c'est-à-dire sur le logiciel Apogée pour accéder aux maquettes des spécialités, à la liste des enseignants et à la liste des salles.

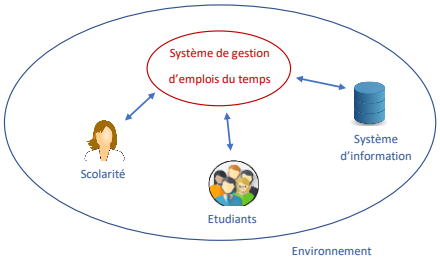
La délimitation environnement-système est la suivante :

Système :
- Partie matérielle : des PCs,
- Partie logiciel : le logiciel à développer,

Scolarité : ensemble du personnel, par spécialité, qui va créer les emplois du temps,

Etudiants : étudiants inscrits dans la formation,

Système d'Information : Apogée.



1°) Les diagrammes des cas d'utilisation UML

C'est un diagramme orienté « fonctionnalités » dont le rôle est d'identifier, formaliser, représenter, et organiser les grandes interactions fonctionnelles entre le système et les acteurs externes. **Ces diagrammes sont centrés sur les Acteurs**.

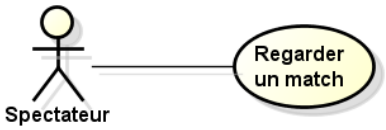
Un diagramme de cas d'utilisation représente un scénario d'interactions possibles entre des Acteurs et le système. Un **Acteur** est un :

- Acteur primaire/principal si c'est lui qui provoque le cas d'utilisation (il se place à gauche du diagramme),

- Acteur secondaire : il participe à un cas d'utilisation mais n'en est pas déclencheur (il se place à droite du diagramme).

Un acteur humain est représenté par un stick man (bonhomme) et un acteur logiciel/matériel par un classificateur UML (cf illustration ci-dessous).

Un diagramme contient plusieurs **cas d'utilisation** représentés par des rond contenant un verbe : un cas d'utilisation identifie une fonctionnalité sans se préoccuper de la façon dont celle-ci va être réalisée.
Un cas d'utilisation est associé à au moins un acteur principal.



L'association peut être stéréotypée si besoin. Par exemple, << participe à>>.

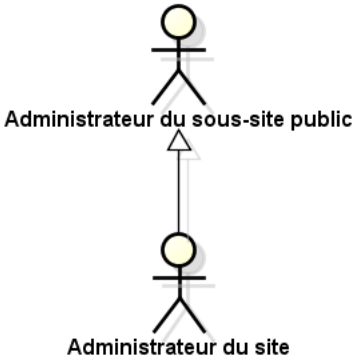
Il nous reste maintenant à préciser quelques notions complémentaires.

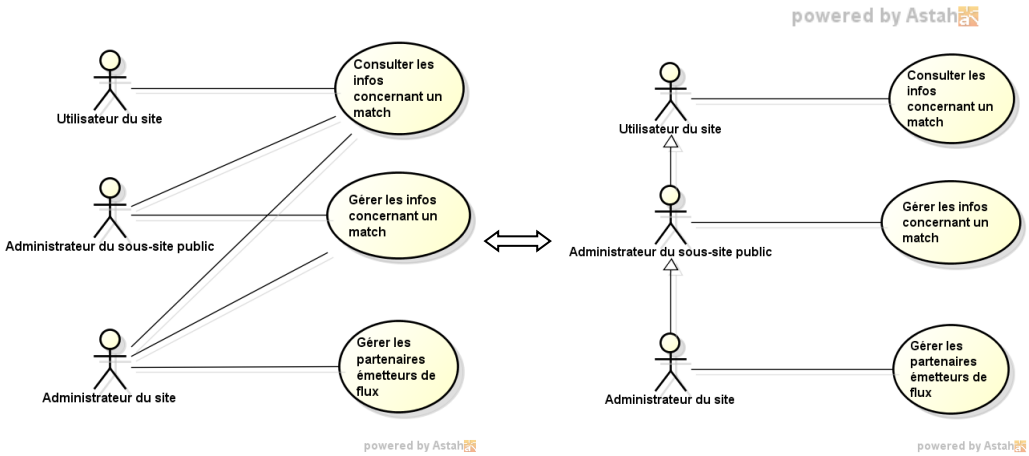
- Héritage entre Acteurs :

  On le représente par une flèche pleine orientée vers l'acteur le plus général.

  Symbolise le fait que l'acteur spécialisé peut faire tout ce que fait l'acteur général… et des choses en plus

  A utiliser quand cela apporte une vraie valeur ajoutée au diagramme, une meilleure lisibilité et/ou structuration !

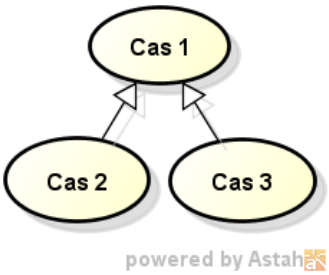




- Généralisation/spécialisation entre cas d'utilisation :

  La **généralisation**/spécialisation est similaire à ce que vous connaissez pour les classes ou pour les acteurs : si le cas A est une généralisation du cas B, B est un cas particulier de A.

  Exemple : on peut généraliser les cas « vérifier les empreintes digitales d'un passager » et « vérifier l'empreinte rétinienne d'un passager » avec le cas « vérifier l'identité d'un passager »



- Représentation : flèche pleine orientée vers le cas le plus général

- Dépendances stéréotypées <<include>> :

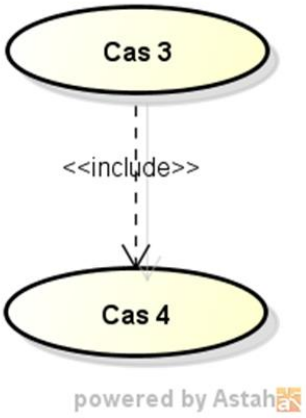
  Elle est utilisée quand un cas d'utilisation doit réaliser, entre autre, un autre cas d'utilisation **de façon obligatoire**.

  Le cas d'utilisation 3 inclut le cas d'utilisation 4, si 4 est nécessairement exécuté quand 3 est exécuté.

  L'inclusion est utilisée pour **factoriser** des parties communes à plusieurs cas d'utilisation,
  ou bien pour **décomposer** un cas d'utilisation complexe en sous-cas plus simples.

  Mais attention à ne pas tomber dans le découpage fonctionnel trop fin…

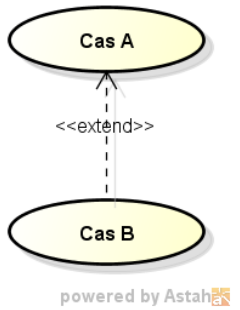
  Rappel : *pas de représentation temporelle* dans un cas d'utilisation ! Donc pas de notion d'enchaînement quand un cas en inclut plusieurs autres.



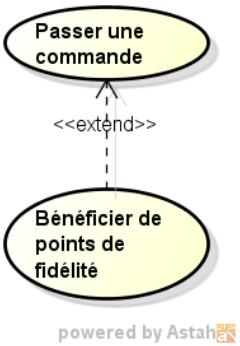
- Dépendances stéréotypées <<extend >>:

Ce type de dépendance est utilisée quand deux cas d'utilisation sont dépendants de telle sorte que la réalisation de l'un puisse entraîner la réalisation de l'autre.

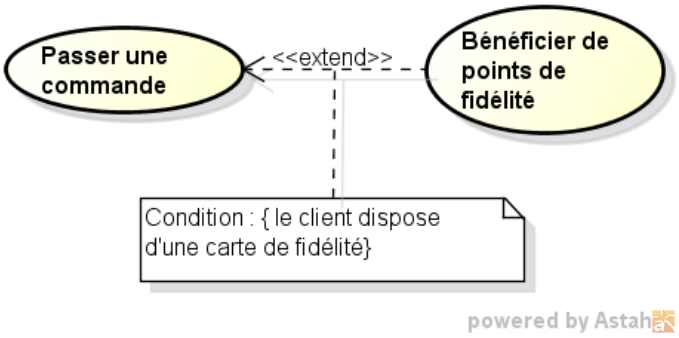
Dit autrement : elle permet de rajouter une fonctionnalité qui n'est pas nécessairement utilisée.
Le cas d'utilisation B **étend** le cas d'utilisation A, si B est **optionnellement** exécuté quand A est exécuté.
La dépendance est représentée de B vers A.



Le cas de base DOIT être un cas **complet** et **autonome**.
Permet d'éviter les extensions d'extensions d'extensions…
… le cas étendu est souvent non pertinent de façon autonome.



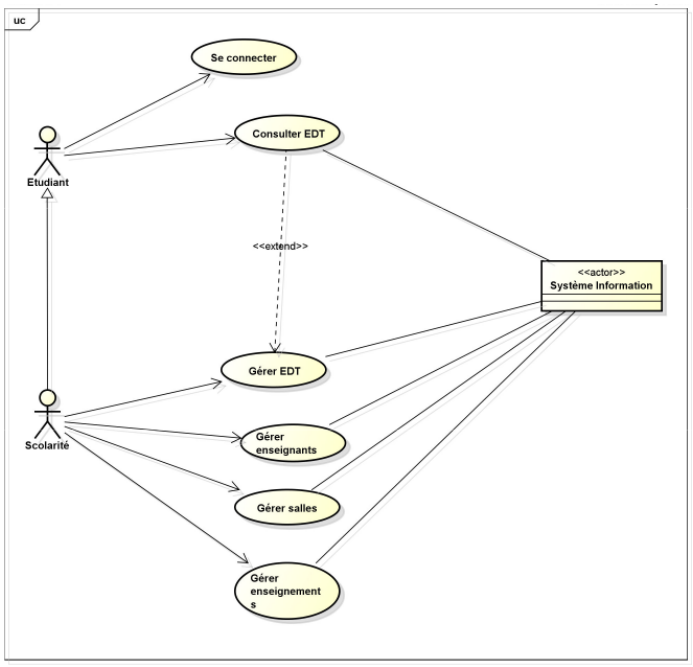
Il est également possible dans tout diagramme UML d'utiliser des notes pour préciser des conditions de réalisation. Par exemple, sur une relation <<extend>> :



Enfin, pour chaque cas d'utilisation il faut également construire un texte explicatif décrivant le cas :

- Nom du cas
- Objectif : c'est la fonction principale du cas, décrite de façon résumée pour comprendre l'idée générale
- Acteurs : liste des acteurs impliqués dans le cas, principaux et secondaires. L'acteur déclencheur et l'acteur cible de l'action sont mis en avant.
- **Une liste de scénarios** :
  - Un **scénario** représente ici une séquence d'échanges acteur/système.
  - On décrira au moins le **scénario nominal,** et éventuellement des scénarios alternatifs et/ou d'exception.
  - Pour chaque scenario : préconditions / déroulement / postconditions

ILLUSTRATION : Reprenons le cas d'application de la gestion des emplois du temps à Polytech Tours. Partant de la délimitation environnement-système, on peut dresser le diagramme des cas d'utilisation suivant : il va synthétiser un ensemble d'interactions entre le système et les scolarités et les étudiants.



Il s'agit d'une illustration, les cas d'utilisation pouvant être précisés, d'autres sans doute rajoutés.

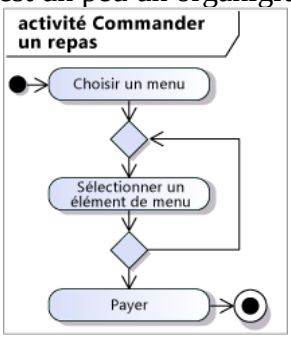
Notez bien que les diagrammes des cas d'utilisation doivent être adaptés à la granularité désirée car ils servent souvent à dialoguer avec le Client. Une tendance naturelle est de faire des cas d'utilisation trop précis… et donc d'avoir des diagrammes trop volumineux.

Notez enfin que, généralement, par cas d'utilisation, on va faire un diagramme d'activité et au moins un diagramme de séquence.

2°) Les diagrammes d'activité UML

2.1) Eléments de base

C'est un diagramme qui met l'accent sur les traitements à un niveau plus fin que le diagramme des cas d'utilisation. Il permet de représenter une activité sous la forme d'un enchaînement de sous-activités : c'est un peu un organigramme.



Dans un tel diagramme on trouve des **actions** qui représentent un traitement élémentaire et ininterruptible. Elle est représentée par un rectangle aux coins arrondis (cf « Choisir un menu » ci-dessus).

Les actions sont reliées entre elles par des **transitions** (considérées comme instantanées) et

qui définissent une chronologie, un flux de contrôle et/ou un flux de données.

En plus des nœuds **actions** on trouve différents nœuds de contrôle :

- Nœuds de début et de fin :

  Le **nœud initial** matérialise le début de l'activité
  - Pas d'arc entrant, un seul arc sortant
  - Une activité peut éventuellement avoir plusieurs nœuds initiaux

  Le **nœud de fin d'activité** met fin à la totalité de l'activité dès qu'un arc entrant l'active.

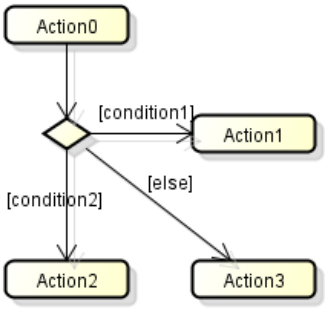
  

  Le **nœud de fin de flux**, optionnel, met fin au flux en cours, mais cela est sans incidence sur les autres flux actifs de l'activité englobante,
  Met fin à une « branche parallèle » parmi plusieurs, le cas échéant

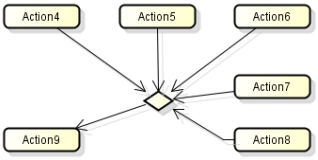
  

- Nœuds de décision / fusion :

  Le **nœud de décision** sert à illustrer une étape de **choix** entre plusieurs flux sortants.
  - Chaque transition sortante est porteuse d'une condition (dite « de garde »)
  - Il convient de veiller à ce que les conditions soient complémentaires…
    - sinon le modèle est mal fait…
    - d'où l'utilisation conseillée d'une branche « else »
  - On n'écrit PAS le test dans le losange ! Mais éventuellement dans une note attachée au nœud décision.

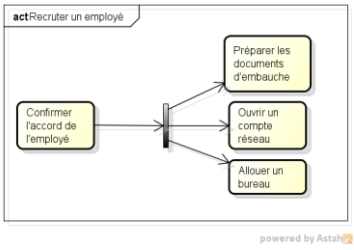
  Le **nœud de fusion** est utilisé pour rassembler plusieurs flux alternatifs en un seul.
  Ce n'est pas un nœud de synchronisation ! (*Cf.* nœud d'union plus loin)

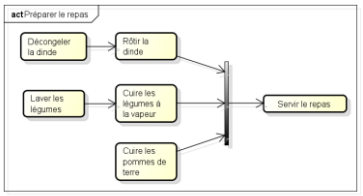
  Ces deux types de nœuds peuvent être fusionnés si besoin.

- Nœuds de bifurcation / union :

Le **nœud de bifurcation** (*fork*) sépare un flux en plusieurs flux concurrents (*i.e.* potentiellement parallèles)



A l'opposé :
Un nœud d'**union** (*join*) synchronise plusieurs flux et les réunit au sein d'un flux unique



L'activité attend que toutes les actions qui précèdent l'union soient terminées pour exécuter l'action qui la suit.

Ces deux types de nœuds peuvent être fusionnés si besoin.

Le diagramme d'activités ne fait pas apparaître que des **flux de contrôle** dans les liens entre actions/sous-activités. Il permet également de modéliser les **flux de données** qui transitent au sein de l'activité.

- Des objets sont produits/créés par des actions/activités puis consommés par d'autres.
- On schématise les **nœuds d'objets** sous la forme de **rectangles :**

Un **nœud d'objet** est caractérisé par :

- Un **type**, éventuellement un nom (on écrira « *nom : type* »)
- Ainsi que - si besoin - un **[état]**, des {contraintes}, un <<stéréotype>>…

Les flux de données peuvent aussi être représentés de façon plus discrète par le biais de « pins » d'entrée/sortie. Un **pin d'entrée** ou de sortie peut être vu comme un nœud d'objet « compacté » sur l'entrée ou la sortie d'une action.

Pour diagramme donné, vous choisissez l'une ou l'autre des représentations, mais pas les deux en même temps !

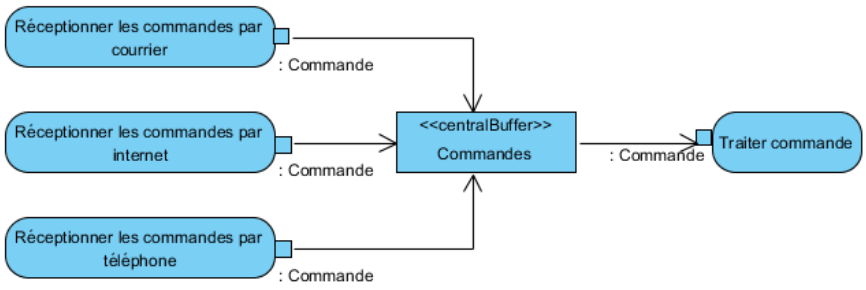
Maintenant, passons à quelques éléments complémentaires !

- Nœuds de tampon central :

  A la différence des autres nœuds d'objet on ne l'utilise pas entre deux nœuds d'actions mais entre deux nœuds objets, standards ou sous forme de pins.

  Comme leur nom l'indique, ils servent de « tampon » entre les actions qui les produisent et/ou les consomment.

  Ils sont schématisés par un nœud objet stéréotypé <<centralBuffer>>

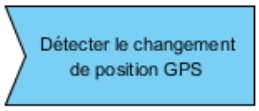


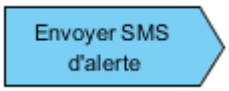
- Nœuds de stockage de données :

  Son rôle est de matérialiser la persistance des données dans un diagramme d'activités.

  A la différence du nœud tampon central « standard » où le flux sortant extrait un élément de la collection, ici le flux sortant duplique l'information qu'il prélève.

- Nœuds d'exécution spécifique :
  - « ***accept event*** »
    Réception d'un événement externe à l'activité décrite
    Souvent utilisé pour de la gestion d'interruptions
  - « ***accept time event*** »
    Cas particulier pour un événement temporel
    (typiquement un temps d'attente)
  - « ***send signal*** »
    Emission d'un signal externe / envoi d'un message.

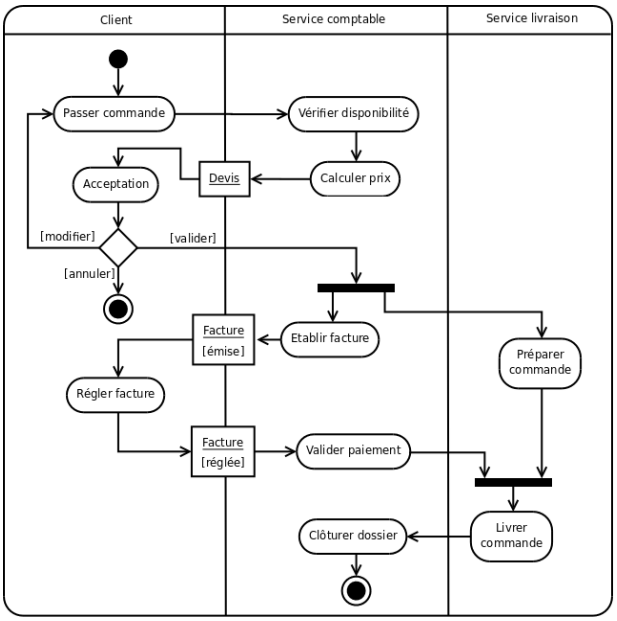




Voyons maintenant un dernier élément, lié à la présentation d'un diagramme d'activité : la **notion de partition**. Il est possible d'organiser les nœuds d'action en effectuant des regroupements logiques. On parle de **partitions** (ou de couloirs, ou lignes d'eau). Ces partitions n'ont pas de signification précise préétablie. Elles peuvent *par exemple* spécifier :

- le service (métier) concerné par les actions,
- la classe en charge de l'implémentation du comportement des nœuds dans la partition,
- …

Il est possible de superposer deux découpages : un horizontal et un vertical.

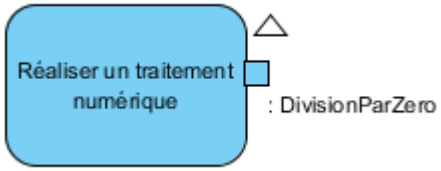


2.2) Exceptions et interruptions

Un diagramme d'activités représente **un ensemble de scénarios proches** pour une réalisation donnée. Parmi ces scenarios figurent les **cas d'exceptions**.

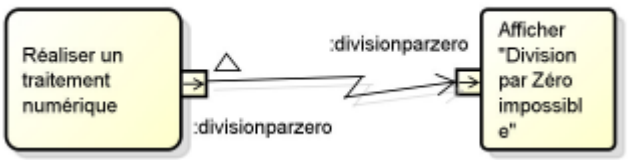
- Une exception est un objet particulier qui est généré automatiquement pour signaler la survenue d'une erreur lors de l'exécution.
- Elle peut être *levée*, *interceptée* ou *propagée*.
- L'activité levant l'exception est définitivement interrompue.

On représente une action pouvant lever une exception par un pin de sortie surmonté d'un petit triangle :



Un **gestionnaire d'exception** est une action dont le rôle est spécifiquement de prendre en charge une catégorie particulière d'exceptions pouvant être levées dans d'autres activités. Ces dernières sont dites **protégées** si elles sont liées à un gestionnaire d'exception pour les types d'exceptions qu'elles peuvent lever.

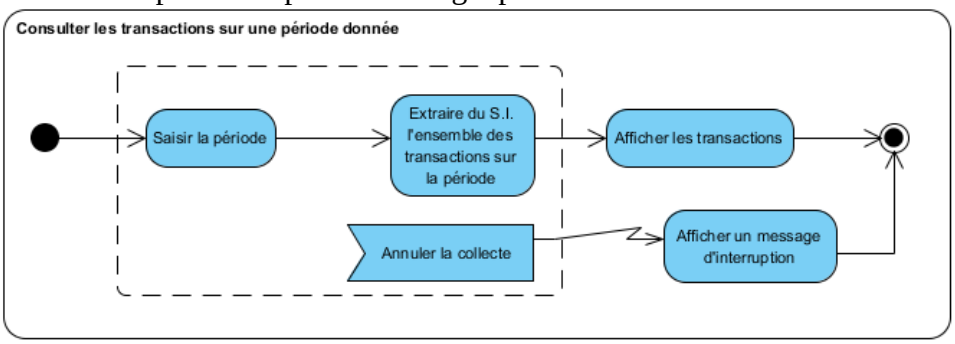
Le gestionnaire d'exception possède un **pin d'entrée** correspondant au type d'exception qu'il sait gérer. Le lien entre l'activité protégée et le gestionnaire d'exception est un arc « éclair » :



Si un gestionnaire d'exception est trouvé et s'exécute, le flux se poursuit à l'issue de l'activité ayant levé l'exception comme si de rien n'était.

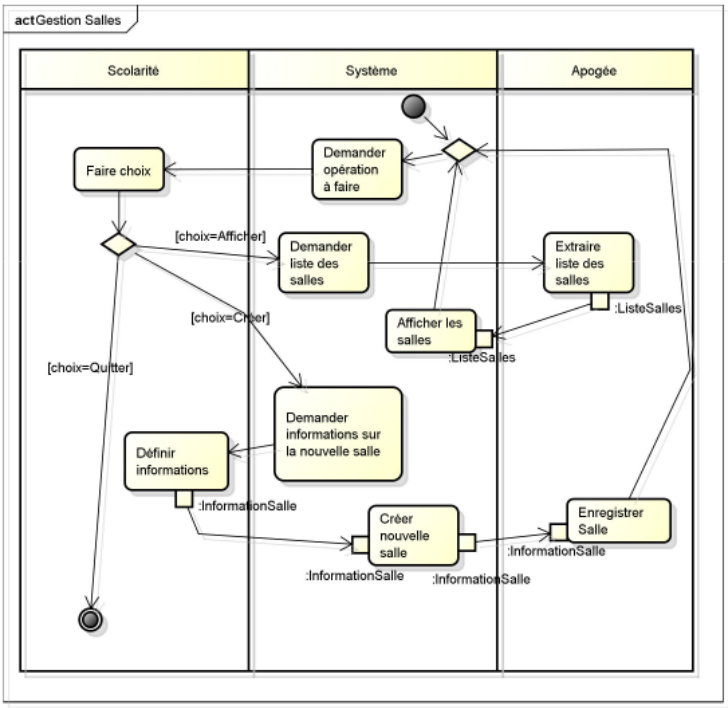
Nous allons maintenant aborder une deuxième notion, celle de **région interruptible**. Une zone du diagramme d'activité qui peut être interrompue définitivement par l'occurrence d'un événement externe est appelée **région interruptible**.

- Toute action en cours dans la zone (y compris les flux multiples) est stoppée et ne sera pas reprise.
- Le flux est transmis au gestionnaire d'interruption via un arc d'interruption
  Même représentation que pour une exception mais sémantique bien différente.
- L'événement déclenchant est matérialisé par un nœud d'exécution spécifiques (pex, accept event).
- La zone est représentée par un rectangle pointillé aux coins arrondis

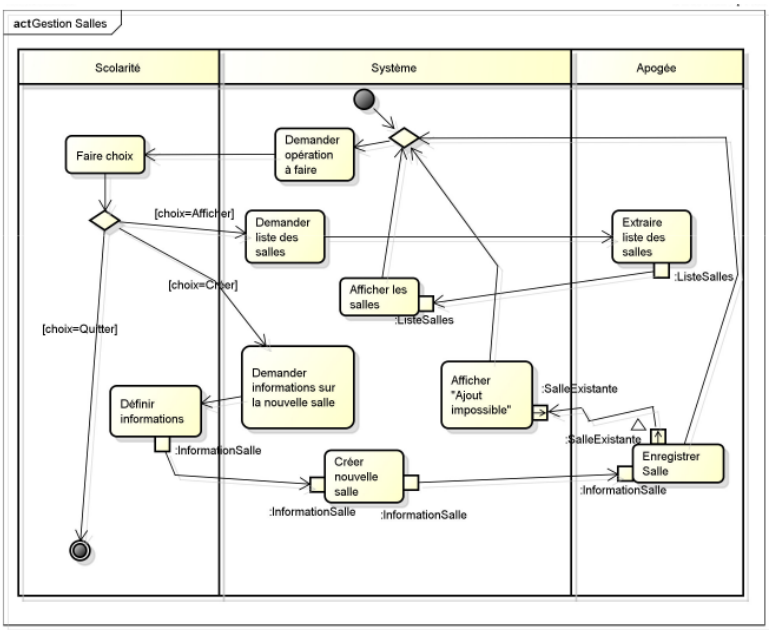


2.3) Cas d'application (continuation)

Continuons avec notre exemple lié à la gestion des emplois du temps à Polytech Tours, et intéressons nous au cas d'utilisation « Gérer les salles ». Nous allons préciser au travers d'un diagramme d'activités les interactions qui se passent quand ce cas d'utilisation est déclenché. Le diagramme ci-dessous, pour des besoins de clarté est limité aux fonctionnalités *Afficher* et *Créer* mais il faudrait bien évidemment, selon les besoins du Client, ajouter également *Modifier* et *Supprimer*.



On pourrait aussi vouloir prendre en compte le fait qu'il n'est pas normal de vouloir enregistrer une salle qui existe déjà. Cela nous donnerait la modification suivante sur le diagramme :



III – Rédiger le Cahier de Spécification du Système

> Comme il a été vu précédemment, ce document sert d'interface entre le Client/Demandeur et l'équipe de développement. Il doit donc être écrit dans un langage compréhensible par les deux parties. C'est là une des difficultés que l'on rencontre dans cette phase : écrire un document qui rapproche deux milieux dont les connaissances sont souvent différentes. En ce qui concerne sa rédaction, on peut proposer le plan inspiré de la norme IEEE std:830 (voir [Jaulent-1992]).

Plan du Cahier de Spécification du Système

I-Introduction

Nature du document, les acteurs mis en jeux, objectifs du document...

II-Contexte de la réalisation

II-1 Objectifs

→Définir en quelques lignes s'il s'agit de faire ou d'améliorer un système... et surtout la nature du système en question.

Ex : Il faut modifier le système Z pour lui permettre de traiter les configurations Y...

II-2 Hypothèses

→Les hypothèses décrivent tous les facteurs susceptibles de remettre en cause tout ou une partie de la réalisation des spécifications ainsi que d'éventuelles solutions de repli.

Ex : Le composant X qui sera utilisé est livré par la société Y, tel algorithme est supposé réalisé par une société en sous-traitance...

II-3 Bases méthodologiques

→Elles précisent quelles procédures, méthodes, outils, normes, langages, règles de programmation devront être utilisés pour mener à bien le projet.

Ex : Application de la norme ISO 9002, utilisation d'un AGL, ...

III-Description générale

III-1 Environnement du projet

→Ce paragraphe situe le projet par rapport à l'environnement dont il dépend et par rapport à d'éventuels projets parallèles. Les limites du projet, ainsi que la place du logiciel dans le système peuvent être visualisées en utilisant le formalisme d'une méthode d'analyse telle que SADT.

Ex : Le projet est traité par les équipes X, Y et Z du département A. Parallèlement à ce projet, l'équipe X est déjà en train de réaliser le projet B ...

III-2 Fonctions générales du système

→Décrire les fonctions utilisateurs du système et les principaux objets que le système manipule. Ceci peut être fait au travers de la présentation de plusieurs modèles : le modèle fonctionnel, le modèle opérateur, le modèle utilisateur occasionnel du système...

III-3 Caractéristiques des utilisateurs

→Identifier les différents types d'utilisateurs du système. Pour chacun on devra préciser les caractéristiques qui affectent l' « interface utilisateur » (menus, commandes textuelles, ...) :

- connaissance ou non de l'informatique,
- expérience de l'application,
- utilisateurs réguliers et/ou occasionnels,
- droits d'accès utilisateurs.

III-4 Configuration du système

→Décrire à l'aide d'un synoptique, la configuration générale du système, en faisant éventuellement référence à un document spécifique.

III-5 Contraintes de développement, d'exploitation et de maintenance

III-5-1 Contraintes de développement

→Préciser les contraintes liées aux :
- matériels,
- langages de programmation imposés ou adoptés,
- logiciels de base,
- environnements nécessaires : simulateurs, outils logiciels,
- algorithmes imposés,
- bibliothèques de programmes imposées,
- protocoles de communication imposés,
- délais de réalisation,

etc ...

III-5-2 Contraintes d'exploitation

→Préciser les contraintes liées aux :
- règles de gestion du système,
- affectations des responsabilités des utilisateurs,
- « planning » d'exploitation,
- problèmes de sauvegardes et de reprises (architecture minimale nécessaire à un fonctionnement dégradé),
- problèmes de sécurité et d'intégrité,
- intervention d'une équipe système,

etc ...

III-5-3 Maintenance et évolution du système

→Préciser les contraintes liées aux procédures de maintenance:
- curative ou corrective,
- adaptative,
- évolutive du système,
- perfective.

IV Description des interfaces externes du logiciel[Grize-1996][Cours SIAD] [Cours Ergo] [Ukelson et al-1993]

IV-1 Interfaces matériel/logiciel

→L'interface matériel/logiciel décrit précisément le matériel informatique et les périphériques, les procédures d'échange d'informations mis en jeu entre eux... On notera donc ici les caractéristiques du matériel qui peuvent avoir une influence sur le logiciel, telles que :
- les normes de communication : protocole d'échange et de raccordement (réseau local ...),
- type de liaison (série, parallèle, synchrone, asynchrone, ...),

etc ...

IV-2 Interfaces homme/machine

→Il faut spécifier les points suivant :
- ergonomie du système : caractéristiques des messages d'erreur, type de navigation dans le logiciel, ...
- description des formes des éditions sur papier et écrans,
- mode d'apprentissage de l'interface éventuellement,
- niveau d'intelligence des interfaces H/M,

etc...

IV-1 Interfaces logiciel/logiciel

→Il faut spécifier les points suivant :

- moyens d'accès à des systèmes de gestion de base de données (bibliothèques mathématiques, graphiques), description de la fréquence des accès, autorisations, ...
- procédures de soumission des travaux à distance (échanges d'informations par téléinformatique),
- procédures d'échange de messages,

etc...

V Description des objets [Cours SADT] [Cours Merise]

Partie la plus importante car il faut identifier les principaux objets (externes) qui composent le système en faisant éventuellement intervenir un dictionnaire des objets. Certains objets, en particulier ceux qui viennent de l'extérieur, sont sujets à des contraintes de représentation. Chaque objet fera l'attention d'un paragraphe V-i.
On présente ici l'arbre hiérarchique des objets.

V-i Définition de l'objet i

V-i-1 L'objet i

→Il faut décrire l'objet en lui même, en tant qu'entité :

- nom de l'objet,
- attributs, caractéristiques de l'objet,
- valeurs de chaque attribut / état de l'objet,
- classe d'objets (niveau d'abstraction supérieur)

Il y a une connotation POO dans cette partie. L'idée est identique à la spécification de classes dans un modèle objet.

V-i-2 Contraintes sur l'objet i

→Quelles sont les contraintes présentes sur les attributs de l'objet, autrement dit sur les états qu'il peut adopter ?

VI Description des fonctions[Cours SADT] [Cours Merise]

Cette partie a comme objectif de décrire l'ensemble des fonctions du système en spécifiant à quel objet elles appartiennent et sur quels objets elles peuvent interagir. Chaque fonction fera l'attention d'un paragraphe VI-i.
On présente ici l'arbre hiérarchique des fonctionnalités.

VI-i Définition de la fonction i

VI-i-1 Identification de la fonction i

→Il faut présenter la fonction :

- nom de la fonction,
- rôle, présentation générale.

VI-i-2 Description de la fonction i

→Il faut décrire précisément le traitement associé à la fonction et à ses interfaces. Il faut pour cela faire référence à la description des objets en précisant si la fonction accède à l'objet, le crée ou le modifie :

- spécification technique détaillée de la fonction,
- objets auxquels elle a accès,
- interfaces avec d'autres fonctions ; références de ces fonctions,
- objets et/ou données en entrée et en sortie.

VI-i-3 Contraintes opérationnelles sur la fonction i

→Il faut spécifier l'ensemble des contraintes qui limitent et régissent le comportement de la fonction.

VII Conditions de fonctionnement

Il faut dans ce paragraphe décrire les dispositions qu'il est nécessaire de prendre en compte dans des conditions particulières de fonctionnement telles que :

- les mécanismes de reprise de fonctionnement en mode dégradé ou minimum,
- les conditions d'appel d'une fonction de traitement d'erreurs, ...
- le rôle de l'opérateur,

etc...

VII-1 Performances

→Préciser en termes mesurables, les spécifications temps réel liées à l'utilisation du système :

- du point de vue de l'utilisateur :
  temps de réponse souhaité, fréquence d'utilisation, temps d'indisponibilité acceptable, etc ...
- du point de vue de l'environnement :
  fréquence moyenne d'acquisition d'états ou de mesures, fréquence maximale, etc ...

VII-2 Capacités

→Décrire les limites des problèmes traitables par le système et les limites des éventuelles extensions comme par exemple :

- nombre max de terminaux,
- nombre max de points d'acquisition,
- nombre max de transactions simultanées de tel type...

etc...

VII-3 Modes de fonctionnement

→Décrire les modes d'exploitation du système tels que :

- la mise sous tension,
- l'arrêt,
- la reprise de secours,
- les modes dégradés,

etc...

VII-4 Contrôlabilité

→Il faut décrire, si elles existent, les spécifications particulières permettant de suivre l'exécution d'un traitement.

VII-5 Sécurité

→Indiquer le niveau de confidentialité du système (contrôle d'accès des utilisateurs, mots clefs, mots de passe, etc...). On en a déjà parlé en partie au paragraphe III-3.

VII-6 Intégrité

→Préciser les protections contre la déconnexion imprévue, les pertes d'information ... et quelles sont les procédures à suivre pour restaurer les données du système. Y-a-t-il des situations non protégées ?

VII-7 Conformité aux standards

→Les références aux standards ou normes (CCITT, ISO, AFNOR, ...) devront être notées.

VII-8 Facteurs de Qualité [Cours CP] [Raffy-1996]

→Il faut ici faire référence à un plan qualité du logiciel si nécessaire (cela dépend de la politique de l'Entreprise).

VIII Justification des choix effectués

Cette rubrique concerne les spécifications envisagées et délaissées, les alternatives possibles, afin de conserver une trace de l'évolution de la pensée.

IX Glossaire

Dans cette partie on doit trouver, classés par ordre alphabétique, les définitions des termes courants utilisés, des termes techniques, abréviation, sigles et symboles employés dans l'ensemble du document.

X Références

Cette dernière partie recense les références techniques sur le projet sur :

- les documents antérieurs à la phase de conception système,
- les documents sur les méthodes,
- les documents bibliographiques (internes et externes),
- les sources d'obtention des documents.

XI Annexes

On doit trouver ici les modèles complets et les méthodes de spécification système sélectionnées (SADT, ...), ainsi que les fonctions utilisées et réutilisables.

XII Index

Cette partie indique les pages où sont traités et mentionnés les sujets et les termes les plus importants du document.

Chapitre 5 : Spécification du Logiciel

I – Introduction à la Spécification d'un Logiciel

L'objectif de la phase de spécification du logiciel est de créer l'architecture du futur logiciel et de décrire le comportement et les liens entre ses différents composants. A cette phase du développement on cherche à rester indépendant des *contraintes technologiques*.

Un exemple de contrainte technologique peut être « *le logiciel va s'interfacer avec le SGBDR Oracle* ». Dans ce cas, en phase de spécification du logiciel, on tiendra compte du fait que le logiciel devra interagir avec Oracle, mais on ne s'intéressera pas à la façon détaillée dont il le fera (Quelles requêtes il doit envoyer à ce SGBDR ? Comment se feront les connexions ? …).

On doit donc faire un travail de design d'architecture. Le design d'architecture (SWEBOCK 2022) implique d'identifier les composants principaux d'un système ; leurs responsabilités, propriétés et interfaces. Cela implique également de définir les relations et interactions entre eux. Les fondements du système sont donc décidés, et tout ce qui relève du détail interne d'implémentation sera décidé plus tard

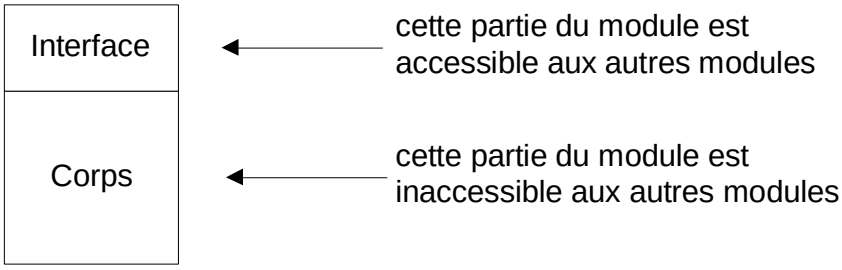
A partir de maintenant nous allons nous placer dans le cadre d'une approche modulaire : elle a l'avantage de pouvoir être implémentée aussi bien dans un langage impératif comme le langage C, que dans un langage orienté objets comme le Java ou le C++. Dans un langage orienté objets, un module sera tout simplement une classe. Dans un langage impératif comme le C, un module sera défini par un fichier .c et .h.

II – Spécifier d'une architecture modulaire

1°) Premiers éléments

Un module est une entité du programme qui est définie par un ensemble d'attributs et de méthodes manipulant ces attributs. Pour prendre un exemple, en langage C on peut faire le lien avec les structures et en C++/Java, ce sont les classes !

Un module peut être découpé, conceptuellement, en deux parties :



On distingue différents types de modules/classes d'un point de vue Génie Logiciel :

- Un module *type abstrait* représente un objet, une entité, du domaine

d'application. Ce type de module est directement induit par le principe d'encapsulation que nous allons détailler ci-dessous.

- Un module *action* représente un ensemble de traitements complexes et pas forcément en lien avec un objet ou entité particulière du domaine d'application.
- Un module *d'Entrée/Sortie*, contient l'ensemble des traitements et objets relatifs aux opérations d'entrée/sortie (sur disque, imprimante, ...) avec l'environnement. Il est nécessaire de les regrouper dans de tels modules car elles dépendent du support matériel : on augmente ainsi la portabilité de l'application et la réutilisabilité des autres modules.

Si on reprend l'illustration développée jusqu'à présent, un module **Enseignant** est un module *type abstrait* alors qu'un module **Calcul_EDT** qui prend en charge le calcul des emplois-du-temps est un module *action*. Il fait un traitement complexe qui nécessite à la fois des enseignants, des salles et des enseignements.

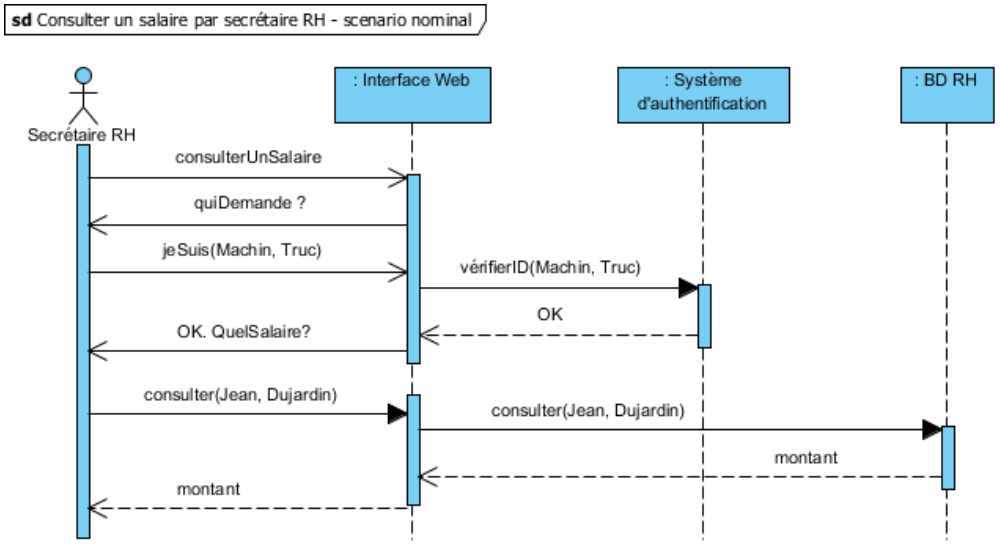
Avant d'aller plus loin, regardons maintenant les mantras du Génie Logiciel donnés en Annexe 1 et leur illustration. Ils vous serviront dès la phase de spécification du logiciel.

NOTE : La « loi de Miller » (résultat de psychologie cognitive) précise que le nombre de concepts, éléments, sur lesquels un homme peut simultanément raisonner est égal à 7 ± 2. Cela implique, que pour un niveau de décomposition d'une abstraction, il est fortement conseillé de ne créer que 7±2 sous abstractions.

2°) Le diagramme de séquence

Le diagramme de séquence est un diagramme UML qui peut être utilisé typiquement à partir de la phase de spécification du logiciel. Il s'agit d'un diagramme comportemental qui met l'accent sur les interactions entre objets (et acteurs) via les échanges d'informations/de données qui circulent entre eux.

C'est le diagramme permettant de visualiser le plus explicitement (parmi les diagrammes UML) **la dimension temporelle** des interactions entre les éléments/composants du système. Voici un tout premier exemple, non commenté, mais qui vous donne la « saveur » de la chose…



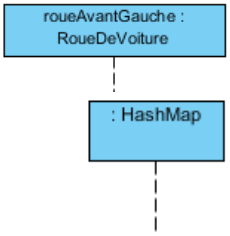
Voyons maintenant les éléments de syntaxe importants :

- Ligne de vie : Une **ligne de vie** représente la participation d'un élément du système à l'interaction décrite dans le diagramme. On parle d'*acteur de l'interaction*. Le nom de la ligne de vie prend globalement la forme suivante :

  nomObjet : nomClasse

  Selon les besoins, l'un ou l'autre peut être omis. Elle peut représenter :

  - Un acteur, au sens cas d'utilisation. *Exemple* : « *Secrétaire RH* »
  - Une instance d'une classe. *Exemple* : « *roueAvantGauche : RoueDeVoiture* »
  - Un élément anonyme d'une classe donnée. *Exemple* : « *: HashMap* »

  La ligne verticale pointillée est partie intégrante de la ligne de vie.
  Une **bande d'activation** (optionnelle) se superpose à la ligne de vie pointillée lorsque l'acteur est **actif** (*cf. exemple Secrétaire RH*)
  Echelle des temps **non homogène** mais **ordre partiel** de haut en bas (et ordre total sur une ligne de vie).

- Les messages : Entre les lignes de vies circulent des **messages**. Il s'agit d'un terme générique indiquant une communication orientée entre les lignes de vies. Provoque une activité dans l'objet destinataire. Il y a plusieurs types de messages.

  - **Les messages d'envoie de signal asynchrones / synchrones.**

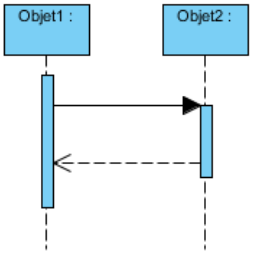
    Certains types de messages sont par nature **asynchrones** *i.e.* qu'une réponse directe n'est pas attendue et l'émetteur n'est pas bloqué en attendant un « retour » de la part du récepteur du message.

    - *Exemple* : envoi d'un signal (*cf.* exemple précédent)
    - Représentation : flèche trait plein extrémité ouverte : (semi-ouverte parfois)

    Au contraire, les messages qui bloquent la ligne de vie source du message sont appelés des messages **synchrones**.

    - Représentation : flèche pleine extrémité fermée :
    - La ligne de vie source est bloquée dans l'état **actif**.
    - Est suivi par un **message de réponse** représenté en pointillé
    - *Exemple de message synchrone* : appel de méthode

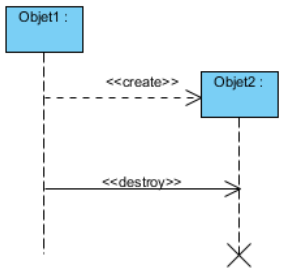
    

  - **La création/destruction d'instances :**

    On matérialise la **création d'un objet** au cours d'un scenario décrit par un diagramme de séquence en utilisant un message pointant sur le sommet d'une nouvelle ligne de vie (l'objet créé)

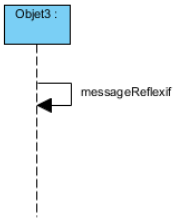
    - Le message peut être stéréotypé <<create>>

    La **destruction d'un objet** sera matérialisée par une croix marquant la fin de la ligne de vie.

- Le message à l'origine de la destruction peut être stéréotypé <<destroy>>

- **Les messages réflexifs :**

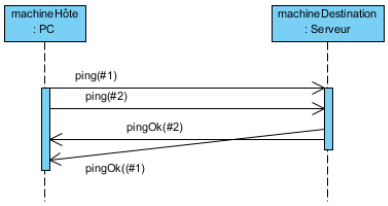


Il est parfois nécessaire de faire apparaître un traitement interne à un acteur/objet.
On utilise pour cela un **message réflexif** :

- *Exemple : appel d'une méthode privée importante.*

- **Les messages rapides/lents:**

UML permet de représenter le fait que certains messages peuvent « prendre du temps» pour être délivrés : on par alors de **messages lents**.



- Par défaut, un message est considéré comme instantané.
- Un message lent est représenté par un trait non horizontal.

Notion de « simultanéité » subjective… adaptée au contexte

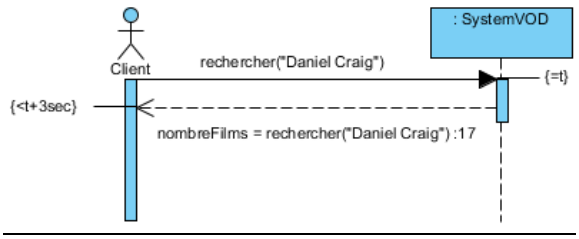
Pour chaque type de message on peut également préciser un certain nombre d'informations :

`[attribut=]nomDuMessage([params])[:valeurDeRetour]`

Tout étant optionnel. Aux messages transitant dans un diagramme de séquences peuvent être associées des **contraintes**

- Sur les paramètres admissibles,
- Sur la durée des différents traitements

Les contraintes sont affichées entre accolades et placées à proximité de l'événement contraint (événement de réception). Exemple : durée max pour une recherche.



Avec les éléments ci-dessous vous avez déjà tout le nécessaire pour commencer à faire des diagrammes de séquence. Voyons maintenant la notion de fragment combiné qui permet d'exprimer un peu plus de chose.

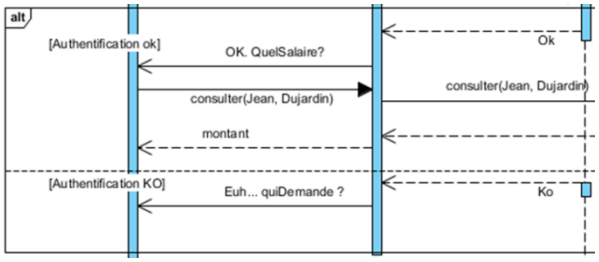
Un **fragment combiné** est une zone de diagramme de séquence caractérisée par des interactions spécifiques dépendantes du type de fragment. Il peut n'impliquer qu'un sous-ensemble des ligne de vies.
Un fragment combiné c'est formellement :

- Un **opérateur** (le type de fragment),
- Des **opérandes** (les sous-fragments)

- Une **sémantique** particulière associée au(x) (sous-)fragment.

Exemple : *le fragment **alt***

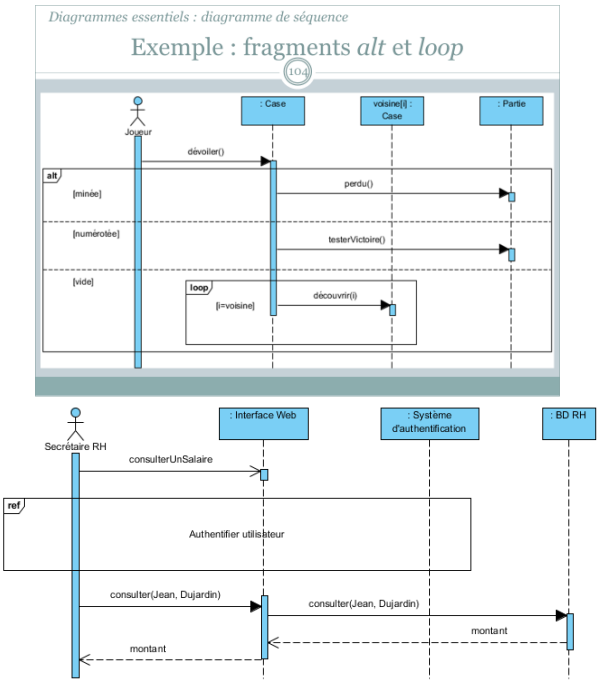
- *opérateur = alt (alternatives)*
- *opérandes = les branches alternatives*
- *sémantique = une seule des branches est exécutée, celle dont la condition de garde est vraie.*



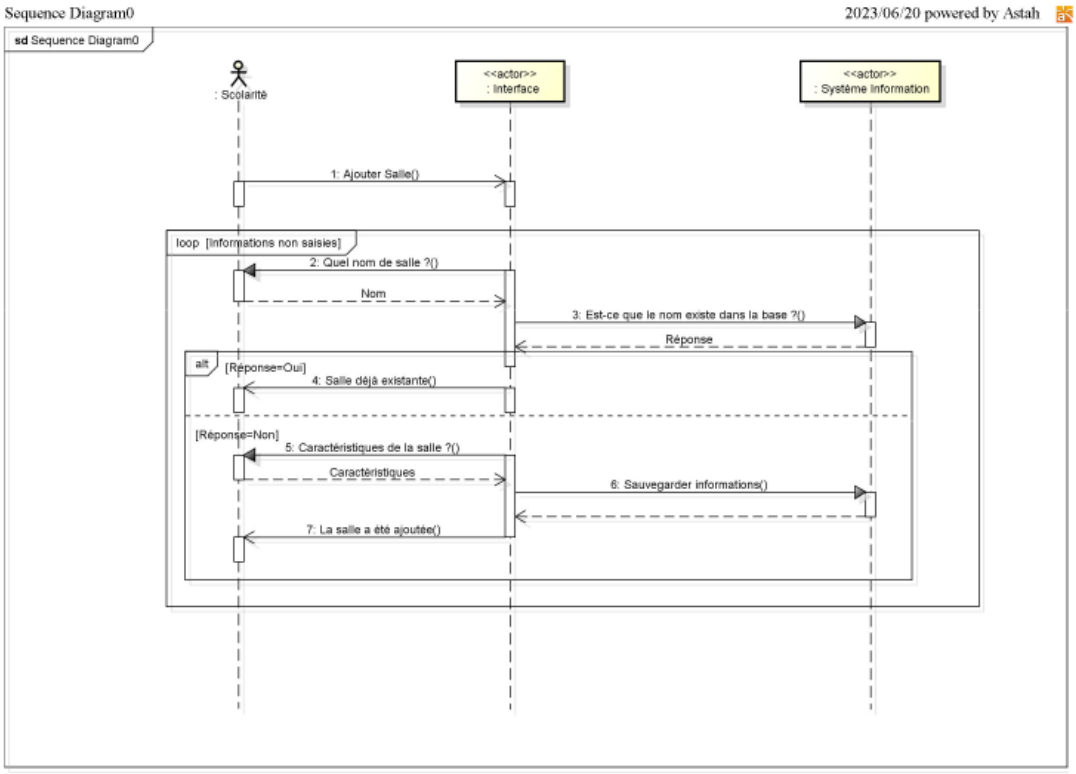
Représentation sous la forme d'un grand rectangle trait plein avec l'opérateur dans un pentagone en haut à gauche et les sous-fragments séparés par un trait pointillé.

Il existe 12 opérateurs de fragments combinés définis depuis UML 2.2 Nous allons n'en voir que quelques-uns...

- Opérateur **alt** (*alternatives*): opérateur conditionnel (type *switch* en C++). Chaque opérande spécifie une condition de garde (*[else]* possible). Un seul sous-fragment est nécessairement exécuté.
- Opérateur **opt** (*option*) : un seul sous-fragment, exécuté de façon optionnelle si la condition de garde est vraie.
- Opérateur **loop** : permet de boucler sur un fragment du diagramme un certain nombre de fois (nombre min et max spécifiés). Une condition de garde peut aussi être testée.
- L'opérateur **ref** (*use*) : permet de faire référence à une interaction dans la définition d'une autre interaction.
- …



Nous allons maintenant illustrer les diagrammes de séquence en reprenant notre exemple lié à la gestion des emplois du temps à Polytech Tours, et intéressons-nous au cas d'utilisation « Gérer les salles ». Nous allons préciser les interactions qui se produisent lorsque l'on chercher à ajouter une salle.



3°) La Démarche

A partir des objets et fonctionnalités définies dans la phase de Spécification du Système et des connaissances que l'on a dans le domaine d'application il faut :

1 - Identifier les objets (objets uniques, classes d'objets identiques) et les actions (actions ne faisant pas d'E/S et actions faisant des E/S) qui sont à prendre en compte dans le logiciel.

2 - Regrouper ces éléments en un petit nombre d'abstractions globales, de façon à définir les modules/classes envisageables. **Bien garder en tête les mantras du Génie Logiciel !**

3 - Analyser les interactions existantes entre les différentes entités en dégageant les actions subies, et provoquées, par chacune d'elles : quelles méthodes de quels modules/classes utilisent quelles méthodes de tel autre module/classe ?

4 – Ne pas hésiter à modifier et faire évoluer votre architecture. Cela nécessite une réflexion approfondie, notamment pour bien prendre en compte les mantras du Génie Logiciel.

Pour chaque module/classe, il vous faudra préciser :

- Son nom, son rôle et son périmètre fonctionnel,
- Les principales structures de données qui le définissent,
- Ses principales méthodes.

Le premier point est notamment important car il vous permettra de savoir, dans votre architecture, ce que doit faire ou ne doit pas faire chaque module/classe.

De même, quand vous analysez votre architecture, gardez un œil critique et notamment sur les points suivants :

- Mon architecture est-elle robuste à des évolutions à venir ?

- Correspond-elle bien au besoin en cours ?
- Bâtir un logiciel selon cette architecture est-il à moindre coût ?
- L'architecture sera-t-elle facilement compréhensible par les développeurs ?

Vous pouvez aussi vous aider de métriques particulières pour évaluer la complexité et la « qualité » de votre architecture (complexité cyclomatique, degré de couplage des modules/classes, degré de compatibilité avec des design patterns existants, …).

 **Dans le cours de Conception Orientée Objets au S6 (DI3), vous verrez le diagramme de classes UML et les design patterns qui sont des approches orientées objets pour représenter et/ou construire des architectures.**

III – Le Cahier de Spécification du Logiciel

Le cahier de spécification du logiciel est une synthèse hiérarchisée et logique qui décrit la solution. Cette présentation suit l'architecture que vous avez définie. On doit trouver également dans ce document la description complète des autres solutions possibles et des choix qui ont conduit à retenir la solution courante.

Plan du Cahier de Spécification du Logiciel

Voici un plan général du *Cahier de Spécification du Logiciel*. Le contenu de ce document dépend énormément des méthodes de modélisation utilisées et des diagrammes construits. Le plan ci-dessous est donc à particulariser en fonction des diagrammes à faire figurer.

1 – Introduction
- Nature du document,
- acteurs mis en jeux,
- objectifs du document, ...

2 - Présentation de l'architecture
- Présentation globale,
- méthodes utilisées,
- découpage sommaire des fonctionnalités en modules,
- Présentation des modules/classes (nom, rôle et périmètre fonctionnel)

3 - Présentation des spécifications des modules/classes
- On décrit plus en détails chaque module/classe, chaque structure de données et les méthodes prévues.

4 - Le modèle comportemental

Présentation des diagrammes de séquence construits pour illustrer la dynamique de l'architecture.

5 - Glossaire

6 - Références

7 - Annexes

8 - Index

Chapitre 6 : Conception du logiciel

I – Objectif

Dans la phase précédente, nous avons établi l'architecture du logiciel. L'objectif de la phase de Conception du logiciel est de transformer cette solution en un programme modulaire « papier » et qui intègre les contraintes technologiques éventuellement définies dans le cahier de spécification du système.

 **Dans cette phase vous allez retrouver toutes les notions que vous avez pu voir dans le cours d'Algorithmique et Structure des Données au S5 (DI3) ou dans la partie algorithmique orientée objets dans le cours de Conception Orientée Objets au S6 (DI3).**

Surtout n'oubliez pas que les mantras donnés dans l'Annexe 1 vous guideront dans la conception de votre architecture et dans l'écriture de vos algorithmes.

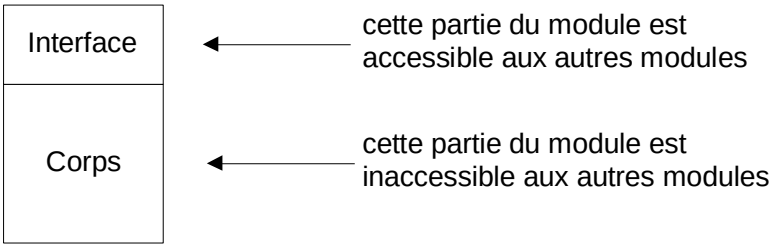
Commençons par voir ce qu'il y a à faire lorsqu'on s'intéresse aux modules/classes.

II – Conception d'une architecture

Nous allons être obligés, dans cette section, de marquer une différence entre une approche modulaire et une approche orientée objets. Nous allons présenter comment implémenter un module (cela est faisable dans un langage impératif même !) et toutes les notions vues seront valables dans le cadre d'un développement orienté objets : c'est simplement la façon dont on implémente ces notions en orienté objets qui diffère. Gardez en tête : un module=une classe en orienté objets.

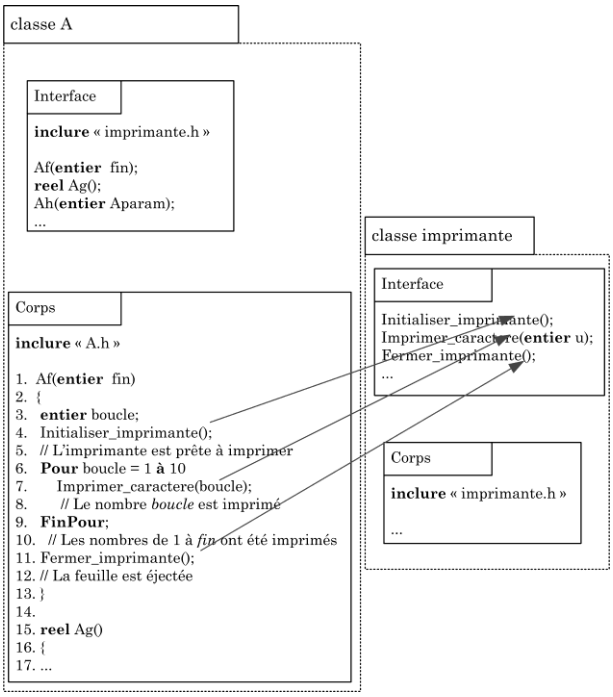
Précisons tout d'abord la notion de module. Comme nous l'avons décrit, **un module est un ensemble d'attributs, de primitives et de sous-modules**. Il doit répondre au principe d'encapsulation, notamment au niveau des données et des traitements (cf Annexe 1).

Conformément au principe d'encapsulation, un module va être constitué d'une *interface* et d'un *corps*.



L'interface est la partie visible du module. Elle contient l'ensemble des types, méthodes et sous-modules accessibles aux autres modules. Il convient de définir avec soin ce qui doit être visible ou non. En effet, la déclaration de la structure d'un type dans l'interface implique que toute unité incluant ce module pourra modifier directement l'objet.

Le corps est la partie cachée du module, où sont décrites toutes les méthodes, types et sous-modules cachés.



La notion d'interface et de corps peut directement être appliquée dans un langage impératif comme le langage C : l'interface correspond à tout ce qui est dans le fichier .h, tandis que le corps correspond à tout ce qui est dans le fichier .c. Facile !
Dans un langage orienté objets les choses sont un peu différentes : vous verrez ça dans vos cours orienté objets !

Pour répondre au principe d'uniformité **PU** (Annexe 1), nous allons définir une norme de rédaction d'un module. Voici ce que nous retiendrons pour l'interface d'un module. Notez, que je vous donne la syntaxe directement dans l'exemple du module **Enseignant** et pour une implémentation en langage C. Le module est incomplet (il manque des méthodes) mais vous comprendrez vite la syntaxe et serez capable de la généraliser à votre langage algorithmique ou à n'importe quel langage impératif.

```c
/*************************************************************
 * MODULE : Enseignant
 * ***********************************************************
 * ROLE : Ce module permet de créer et modifier des informations de
 *        base sur un enseignant.
 * ***********************************************************
 * VERSION : 1.0
 * AUTEUR : Vincent T'KINDT
 * DATE : 20/06/2023
 * ***********************************************************
 * INCLUSIONS EXTERNES :
 */
#include <stdio.h>

/* TYPES :
 *   TEnseignant : une structure contenant les informations propre
 *                 à un enseignant.
 */
struct TEnseignant {
    char* pcENSNom;       // Nom de l'enseignant
    char* pcENSPrenom;     // Prénom de l'enseignant
    unsigned int uiENSAge; // Age de l'enseignant
};

/* VARIABLES :
 *  Nombre d'enseignants créés, un entier non signé
 *
 * METHODES :
 */

/*************************************************************
 * ENSCREER_ENSEIGNANT
 * ***********************************************************
 * Entrée : nom, char *, le nom de l'enseignant
 *          prenom, char *, le prénom de l'enseignant
 *          age, unsigned int, l'age de l'enseignant
 * Nécessite : rien
 * Sortie : un objet enseignant
 * Entraîne : l'objet retourné est initialisé avec les valeurs
 *            passées en paramètre.
 *************************************************************/
extern struct TEnseignant ENSCreer_Enseignant(char* pcNom, char* pcPrenom, unsigned int uiAge);
```

Quelques remarques :

1. La structure a été définie dans l'interface, ce qui n'est pas idéal car l'utilisateur voit comment est implémenté un objet enseignant : néanmoins, pas le choix en langage C.
2. Il y a une variable globale dont on ne connait pas le nom et qui n'est pas déclarée dans l'interface du module : elle ne sera donc pas accessible directement. Il va falloir définir (non fait dans l'exemple) deux accesseurs. Un en lecture qui va faire un return de la variable : on parle alors d'un accesseur direct car il va directement accéder au contenu de la variable. Par exemple,

   ```
   unsigned int Lire_Nombre_Enseignants()
   ```

   Par contre, ce n'est pas dans cet exemple une bonne idée de mettre un accesseur direct en écriture du style :

   ```
   Ecrire_Nombre_Enseignants(unsigned int uiP)
   ```

   En effet, si l'utilisateur peut modifier la variable comme il veut, elle va perdre son sens… c'est plutôt la méthode Créer_enseignant() qui va incrémenter cette variable : cette méthode jouera alors le rôle d'accesseur indirect.
   Pour chaque variable globale, il faudra prévoir au moins un accesseur en lecture et un accesseur en écriture… directs ou indirects ! **Tout ça est la conséquence du principe d'encapsulation.**

Voici maintenant le corps du module correspondant à l'ébauche de l'interface ci-dessus.

```
1  /*************************************************************
2  * MODULE : Enseignant
3  * ***********************************************************
4  * ROLE : Ce module permet de créer et modifier des informations de
5  *        base sur un enseignant.
6  * ***********************************************************
7  * VERSION : 1.0
8  * AUTEUR : Vincent T'KINDT
9  * DATE : 20/06/2023
10 * ***********************************************************
11 * INCLUSIONS EXTERNES :
12 */
13  #include "Enseignant.h"
14
15  /* VARIABLES : */
16  unsigned int uiENSNbEnseignants=0; // Nombre d'enseignants créés
17
18 /*************************************************************
19 * ENSCREER_ENSEIGNANT
20 * ***********************************************************
21 * Entrée : nom, char *, le nom de l'enseignant
22 *          prenom, char *, le prénom de l'enseignant
23 *          age, unsigned int, l'age de l'enseignant
24 * Nécessite : rien
25 * Sortie : un objet enseignant
26 * Entraîne : l'objet retourné est initialisé avec les valeurs
27 *            passées en paramètre.
28 *************************************************************/
29 struct TEnseignant ENSCreer_Enseignant(char* pcNom, char* pcPrenom, unsigned int uiAge)
30 {
31     struct TEnseignant ENSObjet;
32
33     ENSObjet.pcENSNom = pcNom;
34     ENSObjet.pcENSPrenom = pcPrenom;
35     ENSObjet.uiENSAge = uiAge;
36
37     uiENSNbEnseignants++;
38
39     return ENSObjet;
40 }
```

III – Conception des méthodes [Pierra-1991] [Cours AS]

La méthode (appelée aussi parfois fonction ou procédure) est le coeur d'un programme, la brique de base. C'est elle qui définit les traitements sur les objets alors que le module ne permet que de regrouper des éléments. L'obtention des méthodes est l'objet du cours d'algorithmique et structure des données [Cours AS]. Nous ne rappellerons ici que quelques principes de base.

Pour chaque méthode, il faut écrire ses spécifications logiques avant même d'en écrire leur code. C'est une étape très importante qui va impliquer de réfléchir non seulement aux entrées et sorties de la méthode mais également à **ses situations anormales mais prévisibles de fonctionnement**.

Une situation anormale mais prévisible de fonctionnement est une condition d'exécution qui

va faire « planter » la méthode mais qui peut être anticipée. Quelques exemples :

1. La méthode `diviser(numerateur, denominateur)` ne peut pas accepter en second paramètre la valeur 0.
2. La méthode `supprimer(Liste, position)`, qui supprimer un élément de la `liste` dans la `position` donnée, ne peut pas s'exécuter correctement si la position est hors liste.

Il y a deux façons de gérer une telle situation : soit on gère en précondition, soit on gère en postcondition. Les **préconditions** précisent les conditions que doit respecter l'utilisateur de la méthode pour pouvoir l'utiliser correctement. Les **postconditions** sont les engagements de la méthode à remplir une tâche. Dans le premier cas précédent, une précondition serait : « denominateur ne doit pas être nul ». Si on avait géré en postcondition nous n'aurions rien imposé à l'utilisateur mais la méthode aurait testé si denominateur=0 et renvoyé un code d'erreur dans ce cas.

Au niveau de l'écriture, et comme souligné dans les exemples de code donnés dans la section précédente, avant chaque méthode, dans l'interface et le corps du module, on indiquera un cartouche de commentaire correspondant à la spécification logique. Dans l'exemple, de la méthode `Créer_Enseignant`, nous avions :

```
27  /*************************************************************
28  * CREER_ENSEIGNANT
29  * ************************************************************
30  * Entrée : nom, char *, le nom de l'enseignant
31  *          prenom, char *, le prénom de l'enseignant
32  *          age, unsigned int, l'age de l'enseignant
33  * Nécessite : rien
34  * Sortie : un objet enseignant
35  * Entraîne : l'objet retourné est initialisé avec les valeurs
36  *            passées en paramètre.
37  *************************************************************/
```

Revenons un peu sur ces histoires de précondition vs postcondition et regardons le principe que l'on appelle : la dualité précondition/postcondition.

Si l'utilisation d'une précondition permet souvent de définir un domaine d'entrée qui assure la possibilité de réaliser la méthode, encore faut-il que l'appartenance à ce domaine soit raisonnablement vérifiable par l'action utilisatrice. Ceci n'est pas le cas :

- si l'évaluation de la précondition est aussi complexe que l'algorithme lui-même,
- si la possibilité d'exécution de la méthode dépend de conditions extérieures, par exemple d'une entrée provenant de l'environnement.

Dans le cas ou une précondition n'est pas raisonnablement vérifiable, il faut prévoir les différentes situations qui peuvent en découler. On est donc amené à spécifier une postcondition sous forme de disjonction de cas. Ceux qui correspondent à un traitement incorrect, anormal, sont gérés par un mécanisme d'exception.

EXEMPLE : Dans le cas de la réalisation d'un algorithme de résolution d'un système d'équations linéaires, la possibilité d'exécution d'une telle action nécessite de vérifier que la matrice est régulière, ce qui est d'une complexité voisine de la recherche de la solution, si celle-ci existe. Pour être utilisable, l'algorithme devra donc impérativement admettre en entrée n'importe quel système linéaire et fournir en sortie :

- soit la solution du système,

- soit une exception précisant qu'une telle solution n'existe pas.

La spécification fonctionnelle de cette action est :
Résoudre_système_linéaire
E/ matrice A(100,100), vecteur B(100)
nécessite
S/ vecteur X(100) ; logique : EXCEPTION
entraîne {(X t.q. AX=B et non EXCEPTION) ou EXCEPTION}
{EXCEPTION pivot_nul : correspond au cas où apparaît un pivot $<10^{-5}$}

Il y a enfin un principe de programmation qu'il est important de garder en tête, c'est celui de la **compatibilité ascendante** : quand vous programmez, votre code va être amené à évoluer par la suite. Partiellement ou totalement, ce sont sans doute d'autres programmeur qui s'y colleront. Vous devez leur faciliter la tâche. Pour cela, vous pouvez simplement déjà faire en sorte que vos modules/classes offrent vraiment des services indépendant de leur implémentation. Prenons l'exemple d'un module **Liste** qui permet de gérer une liste d'entiers. Dans votre version du module, vous avez fait le choix d'utiliser des listes chaînées pour implémenter la liste. Quelles méthodes allez vous proposer pour ajouter un élément à la liste ? Deux choix possibles :

```
LISAjouter_Element(Liste LISRoot, Liste * pLISElement)
```

ou

```
LISAjouter_Element(Liste LISRoot, int iElement)
```

Le premier choix part du principe que vous n'aurez qu'à chaîner le paramètre `pLISElement` à la fin de la liste chaînée pointée par `LISRoot` (par exemple). Dans le second cas, c'est la méthode qui va créer la cellule `Liste * pLISElement` qui contiendra `iElement` et qui sera ajouté à la fin.
Laquelle version favorise le plus la compatibilité ascendante ? La seconde ! Dans le cas de la première implémentation, si vous décidez de ne plus coder par des listes chaînées vous allez être obligé de changer l'interface de la méthode pour vous ramener au second cas… et il faudra également retoucher à toutes les parties du programme qui faisaient appel à la méthode. On dit que le reste du programme ne sera plus compatible avec la v2 de votre module.

IV – La démarche

La démarche est relativement simple : il faut pour chaque module/classe écrire l'interface puis le corps. Lors de l'écriture des algorithmes des méthodes, il faudra au préalable (lors de l'écriture des spécifications logiques) identifier les situations anormales mais prévisibles de fonctionnement… et décider de comment les gérer.

Ce dernier point est notamment intimement lié à la règle bonne pratique de protection modulaire **PRM** (cf Annexe 1).

Notez également que d'un point de vue pragmatique on n'écrira pas TOUS les algorithmes du logiciel mais seulement ceux qui sont les plus complexes et qui nécessitent une réflexion préalable. Les autres, beaucoup plus simples, pourront être conçus en même temps qu'ils sont codés.

V – Le Cahier de Conception du logiciel

Le Cahier de Conception du logiciel doit rendre compte de la solution proposée dans son intégralité. On doit donc y faire figurer, non seulement le détail de la solution en termes informatiques, mais aussi des différentes méthodes implémentées pour résoudre le problème posé.

Plan du Cahier de Conception du Logiciel

1 – Introduction

- Nature du document,
- acteurs mis en jeux,
- objectifs du document, ...

2 - Présentation des méthodes de résolution

Dans ce chapitre on décrit de manière théorique l'ensemble des méthodes de résolution qui ont été utilisées en justifiant éventuellement les choix effectués (notamment lorsque plusieurs méthodes pour résoudre un même problème sont disponibles).

3 - Présentation de la solution finale

On présente dans ce chapitre la décomposition modulaire finale. Dans une première partie on décrit l'arbre ordonné présentant la hiérarchie des modules et actions abstraites. Dans une seconde partie on détaille chaque module (interface et corps) selon un plan qui suit la décomposition modulaire. Pour un module donné on présente (selon les référentiels vus en cours de Génie Logiciel) :

- Les types définis dans le module (à l'aide des Types Abstraits de Données, par exemple),
- Les actions/fonctions définies dans le module et pour chacune d'elle ses spécifications logiques. Dans la présentation du corps du module, pour chaque action/fonction on donnera son pseudo-code en langage algorithmique.

4 - Glossaire

5 - Références

6 - Annexes

7 - Index

Chapitre 7 : Codage

I – Présentation

L'objectif de cette phase est de transformer tout le travail des deux phases précédentes en un programme informatique. Bien sûr il faudra toujours suivre un certain nombre de règles pour que le code produit soit de qualité. De même, ce qu'il faut absolument éviter est de se retrouver à concevoir un algorithme en même temps qu'on le code : c'est le meilleur moyen de créer des bugs ou d'introduire des problèmes de conception.

ILLUSTRATION : Quelques chiffres sur les enjeux de la phase de codage :

- Une étude de l'US Air Force en 1972 estime le coût d'écriture d'une instruction à 75 $ et le coût de mise au point et de correction d'une instruction à 4000 $ [Pierra-1991].
- Une étude de S.F Zeigler en 1995 (thèse de doctorat) montre que le coût d'écriture d'un programme en langage C est deux fois plus couteux que s'il avait été écrit en langage ADA.

Nous allons dans le reste de ce chapitre mettre en avant quelques éléments de bonne pratique à prendre en compte lors de l'écriture du code.

II – Eléments de bonne pratique

Déjà, en préambule, relisez donc bien l'Annexe 1 et les règles qui impactent la phase de codage.

1°) Norme d'écriture du code

Le principe d'uniformité **PU** (cf Annexe 1) va impliquer que l'écriture de votre code suive un standard de rédaction. Nous avons commencé à en voir un dans la phase de Conception du Logiciel, concernant les modules/classes et les en-têtes de méthodes.

Nous allons aller plus loin en imposant également des conventions de nommage sur les types, variables et méthodes.

Nous nommerons tous **les types** en commençant par la lettre T en majuscule et suivi du nom en minuscule, par exemple, Ttableau, Tchaine,...

Concernant **les noms de fonctions** nous utiliserons les trigrammes pour les fonctions appartenant à des modules/classes (méthodes). Le trigramme sera généralement constitué par les trois premières lettres du nom du module/classe. Lorsque la fonction n'appartient pas à une méthode/classe (la règle pour les structures est identique au cas des classes) alors on utilisera simplement son nom en le faisant commencer par une majuscule.

Le cas des **variables** est un peu plus complexe. Chaque nom de variable sera précédé d'une ou plusieurs lettres selon le cas de figure. Si cette variable est un *pointeur* alors son nom commencera par la lettre p suivi de lettres pour préciser le type sur lequel elle pointe. On utilisera le tableau suivant pour les correspondances : celui-ci est donné dans le cas des

langages C/C++ mais vous pourrez facilement l'adapter à un autre langage !

Lettre	Type de base
c	char
i	int
ui	unsigned int
f	float
d	double
b	bool

Par exemple, une variable de nom `boucle`, de type i**nt**, sera nommée `iBoucle`. Une variable de nom `ligne` et de type pointeur sur un **caractère** sera nommée `pcLigne`.
Si la variable est un attribut d'une classe on utilisera le trigramme associé à la classe (cf. ci-dessus et le nommage des fonctions). Ainsi, si la variable `pcLigne` appartient à la classe `Cliste_generique`, son nom devient `pcLIGligne`.

Regardez à nouveau les bouts de code C qui étaient donnés à titre d'exemple dans le chapitre précédent et vous verrez que nous avions déjà utilisé cette convention de nommage !

Bien évidemment, la proposition faite dans ce cours nous est propre et d'une entreprise à l'autre les standards peuvent changer !

2°) Compatibilité ascendante, factorisation and co

La règle de bonne pratique **COM** implique notamment qu'il faut se plier à l'exercice de la factorisation de code. Nous l'avons largement abordé dans l'Annexe 1 : pas la peine de revenir dessus !

Gardez en tête de faire simple ! **Le meilleur code est le plus simple**. Globalement, multiplier les méthodes pour qu'elles soient de petite taille est un bien meilleur parti que de faire des méthodes volumineuses et difficilement maintenables.

Pensez également à commenter vos codes ! Un bon programme est aussi un programme qui se lit comme un roman.

Appliquez toujours la règle du typage au plus juste : si une variable prend des valeurs entières non négative, ne la typez pas int en langage C par exemple. Il faut la typer unsigned int.

Bien garder également en tête le principe de **compatibilité ascendante** introduit en phase de conception du logiciel.

III – Le Cahier de Codage

Ce document doit présenter le programme obtenu, c'est-à-dire non seulement le listing mais aussi toutes les adaptations (à limiter) faites lors du codage vis-à-vis de ce qui avait été conçu dans la phase précédente.

Plan du Cahier de Codage

1 – Introduction
- Nature du document,
- acteurs mis en jeux,
- objectifs du document,
- langage de programmation utilisé, ...

2 - Présentation des conventions adoptées
- Décrire l'ensemble des normes de présentation adoptées (standardisation de la présentation des modules, normalisation du nommage des types, variables, fonctions, … et normalisation du langage de programmation).

3 - Présentation du codage

On présente dans ce chapitre le codage de chacun des éléments (actions/fonctions, modules, ...) de l'arbre ordonné défini dans la phase de conception du logiciel. Le découpage en paragraphes doit suivre l'ordre partiel de codage.

4 - Présentation des problèmes rencontrés

Dans ce chapitre on expose l'ensemble des problèmes rencontrés (contraintes liées au langage utilisé, ...) lors du codage et des choix effectués pour les contourner.

5 - Glossaire
6 - Références
7 - Annexes
8 - Index

Chapitre 8 : Tests unitaires des composants

I – Objectifs

Les composants obtenus à la phase précédente doivent être validés au niveau interne, de façon à certifier la phase de conception du logiciel. C'est l'objectif de cette phase. Les tests unitaires sont généralement divisés en trois étapes :

- la revue de code,
- les tests fonctionnels (boîte noire),
- les tests structurels (boîte blanche).

NOTE : l'objectif des tests unitaires n'est pas de prouver « qu'un programme est juste ». Ils peuvent par contre : mettre en évidence les erreurs résiduelles et conforter de façon importante la conviction que l'on a que le programme fonctionne effectivement en conformité avec sa conception.

II – La revue de code

L'essentiel de la validation d'un programme (et de la découverte d'erreurs éventuelles) ne réside pas dans des tests avec des jeux de données mais dans la *vérification intellectuelle, par un autre intervenant, de la correction du code écrit*. Cette vérification s'appelle la revue de code et elle met en jeu un *auteur*, celui qui a réalisé le code, et un *lecteur*, celui qui va lire le code.

Lorsque l'on vérifie du code, la première chose est de travailler module par module (ou classe par classe). Il faut bien vérifier que :

1. chaque module/classe vérifie les standards de rédaction en vigueur,
2. les types et variables sont bien déclarées, au juste type,
3. pour chaque méthode, les spécifications logiques sont compréhensibles et correspondent au traitement réalisé par la méthode,
4. le code de chaque méthode est compréhensible, bien documenté, simple et respecte les règles de programmation établies (convention de nommage, …).

Les erreurs et les anomalies éventuelles sont notées par le lecteur. Le rapport de revue est transmis à l'auteur qui effectuera les corrections.

ILLUSTRATION : Une étude présentée dans [Lleres et al-1989] montre qu'¼ d'heure de revue de code trouve autant d'erreurs que 4 heures de tests à l'aide d'un débugueur.

III – Les tests fonctionnels (tests en boîte noire)

L'objectif des tests fonctionnels est d'exécuter de façon exhaustive chaque action de sorte à vérifier, qu'en respectant les préconditions, qu'elle ne plante pas et se déroule « normalement ».

Pour cela il va falloir identifier :

- les différentes classes de valeurs d'entrée correspondant aux différents comportements possibles du programme,
- les différentes classes de valeurs d'entrée susceptibles, d'après l'expérience du concepteur du test, d'être à l'origine d'erreur (mise en évidence d'erreurs).

Nous allons voir une méthode exhaustive pour spécifier et réaliser ces tests : il s'agit de la méthode de [Ostrand et al-1988].
N'oubliez pas que maintenant les IDE intègrent des outils pour réaliser ce qu'ils appellent des tests unitaires et qui sont en fait les tests fonctionnels. **Vous pouvez donc tout à fait écrire les tests fonctionnels en même temps que vous développez des modules/classes !**
Cela a aussi le mérite de détecter ce que l'on appelle les *régressions de code*.

Pour spécifier un test fonctionnel pour une méthode il faut définir les classes d'équivalences de ses paramètres d'entrée. *Une classe d'équivalence est l'ensemble des valeurs des paramètres d'entrée qui provoquent le même comportement de la méthode*. Ainsi deux jeux de données qui appartiennent à la même classe sont jugés équivalents puisque l'action s'exécuter de la même façon pour l'un et l'autre de ces deux jeux. Une fois le concept de classe d'équivalence compris, il ne reste plus qu'à présenter la démarche de spécification d'un test fonctionnel.

Pour une méthode donnée, celle-ci est composée des étapes :

1 - Identification des objets et variables impactant

Il faut identifier les objets/variables qui influent sur l'exécution de chaque fonction en distinguant les paramètres de la méthode et les éléments externes (variables globales, attributs statiques, …),

2 - Analyse,

Il faut, pour chacun des objets identifiés dans l'étape précédente, analyser les différents *types d'informations* (signe, valeur, ...) qui pourraient influencer le déroulement de l'action,

3 - Partitionner

Il faut découper ces *types d'informations* en domaines moyens et en domaines limites à partir des spécifications de la méthode :

* les domaines moyens correspondent aux différents modes de fonctionnement normaux,
* les domaines limites sont susceptibles d'être à l'origine d'erreurs : cas limites, anomalies, configurations particulières de l'environnement.

4 - Spécifier

On spécifie chaque test à réaliser en exprimant, sous forme de produit cartésien de domaines, la classe d'équivalence qui lui est associée.

EXEMPLE : Prenons l'exemple de la fonction suivante en langage C.

```
1     #include <iostream>
2
3     /*************************************************************
4     * FACTORIEL
5     * ************************************************************
6     * Entrée : iRacine, int, valeur du factoriel à calculer
7     * Nécessite : rien
8     * Sortie : la valeur de iRacine!
9     * Entraîne : le factoriel de iRacine est calculé
10    *************************************************************/
11    double Factoriel(int iRacine)
12    {
13        unsigned int uiBoucle;
14        double dFactoriel = 1;
15
16        for (uiBoucle = 0; uiBoucle <= iRacine; uiBoucle++)
17            dFactoriel *= (double)uiBoucle;
18
19        return dFactoriel;
20    }
21
```

Appliquons la démarche de Ostrand. Voici les différentes étapes.
1 - Le déroulement de la fonction dépend uniquement du paramètre : `iRacine`.
2 – Puisque le paramètre est typé `int`, les types d'informations significatifs sont : la valeur et le signe.
3 - Choix des domaines:
Valeur :
domaines moyens : (A) 0 ≤ `iRacine` ≤ 33
domaines limites : (B) 34 ≤ `iRacine`
⇨ Le choix de ces domaines est basé sur le fait qu'un `double` ne peut pas dépasser 10^{37} et 34 est la première valeur dont le factoriel dépasse ce seuil.
Signe :
domaines moyens : (C) `iRacine` est positif
domaines limites : (D) `iRacine` est négatif
⇨ Le choix de ces domaines est basé sur le fait que `iRacine` peut prendre des valeurs négatives, ce qui peut paraître étrange pour une racine d'un factoriel. Mais rien ne l'interdit dans les spécifications.
4 - Spécification des tests :
On a ici 3 classes d'équivalences : (A+C), (B+C), (D).

On va donc écrire 3 tests, chacun appartenant à une classe ci-dessus. Par exemple, on pourra tester :

- (A+C) : `iRacine` = 10 et vérifier que le résultat est bien 3 628 800.
- (B+C) : `iRacine` = 34 et vérifier si le résultat est correct ou pas.
- (D) : `iRacine` = -10 et voir ce qui se passe lors de l'exécution.

L'analyse de cet exemple et de ces tests va vraisemblablement faire ressortir que :

- Le paramètre n'est pas bien typé : il aurait dû être typé `unsigned int` pour interdire de passer une valeur négative.
- Le programmeur a également oublié de gérer la situation anormale mais prévisible de fonctionnement correspondant au cas ou 34 ≤ `iRacine`. Là, on pourrait gérer en précondition ou postcondition : en tout cas, il faut modifier le code et sa spécification logique.

IV – Les tests structurels (tests en boîte blanche)

Malgré l'utilisation de méthodes systématiques, telles que celles que nous avons étudiées dans la section précédente, la conception de tests fonctionnels reste une activité empirique. Quels que soient les critères utilisés pour choisir les tests à effectuer, on ne peut être assuré ni que la validité des tests entraînera la validité du programme ni que toutes les erreurs éventuelles aient été détectées. L'objectif des tests structurels est de vérifier que toutes les parties d'un programme ont été suffisamment testées.

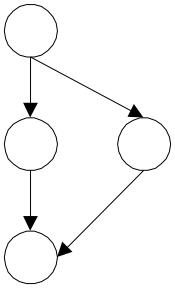
Pour cela on définit la notion de graphe de contrôle d'une action :

- chaque instruction susceptible d'interrompre le déroulement séquentiel du programme (départ d'un « saut de contrôle ») est représentée par un sommet,
- chaque instruction susceptible d'être atteinte sans que l'instruction qui la précède ait été exécutée (arrivée d'un « saut de contrôle ») est représentée par un sommet,
- chaque possibilité de passage d'un sommet à l'autre, soit par un saut, soit par une séquence d'instructions ne comportant ni départ ni arrivée de saut (« bloc séquentiel ») est représentée par une arête.

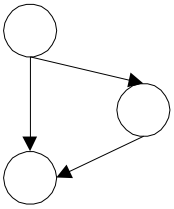
NOTE : chaque sommet du graphe est numéroté et les numéros sont reportés dans l'algorithme correspondant (pas le code).

Les principales structures de contrôle sont représentées de la manière suivante :

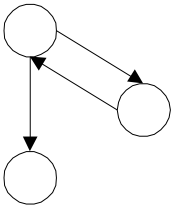
Si... Alors ..Sinon



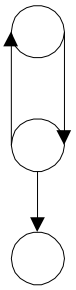
Si... Alors



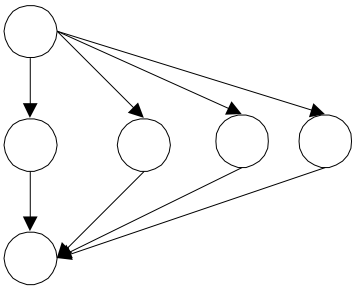
Tant que ...Faire



Faire... Jusqu'à



Selon cas ...

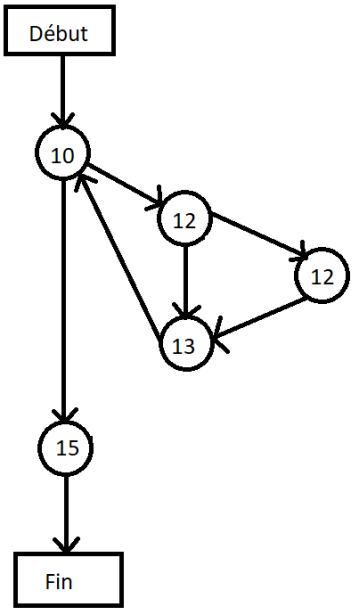


EXEMPLE :

Prenons la fonction suivante écrite en langage C qui teste si un élément appartient à une liste.

```
4   bool IsElement(int * piListe, unsigned int uiTaille, int iElement)
5   {
6       unsigned int uiBoucle;
7       bool bResultat = false;
8
9       uiBoucle = 0;
10      while (uiBoucle < uiTaille && bResultat==false)
11      {
12          if (piListe[uiBoucle] == iElement) bResultat = true;
13          uiBoucle++;
14      }
15      return bResultat;
16  }
```

Sa traduction sous forme de graphe de contrôle est la suivante :



On peut définir sur le graphe de contrôle plusieurs exigences de couverture. Une exigence minimale implique que toutes les branches du graphe aient été parcourues lors des tests fonctionnels. Une exigence maximale, hors d'atteinte dès que le programme contient une structure répétitive, implique que tous les chemins qui correspondent à des exécutions possibles aient été parcourus lors des tests fonctionnels. Etant donné qu'il est impossible de garantir la couverture maximale et que la couverture minimale ne permet pas de tester complètement une action, trois règles de couverture doivent être vérifiées.

Règle de couverture de l'alternative

L'ensemble des jeux de test doit permettre au minimum, si cela est possible, d'exécuter chacune des branches définies par l'alternative.

Règle de couverture de la répétitive

L'ensemble des jeux de test doit permettre, si cela est possible, d'exécuter chaque corps de boucle 0 fois, 1 fois, le nombre maximum de fois.

Si le nombre maximum de fois est difficile ou impossible à atteindre on cherchera à parcourir le corps de boucle 0 fois, 1 fois et 2 fois.

Règle de couverture des prédicats

L'ensemble des jeux de test doit assurer, si cela est possible, la couverture totale des prédicats (chaque terme a été expérimenté pour un jeu de données du domaine

correspondant à la valeur vrai (idem pour faux) et pour un jeu de données appartenant à la frontière entre les deux domaines si celle-ci est non vide).

S'il est relativement facile, soit en utilisant un outil préexistant soit en instrumentant le programme (ajout de composants/mouchards temporaires dans le code), de vérifier a posteriori que chaque instruction ou chaque branche d'un programme a été atteinte lors d'une exécution, il n'existe par contre aucune méthode *automatique* permettant de décider si une branche est atteignable ni, a fortiori de déterminer le domaine d'entrée qui lui correspond. Lorsque les exigences de couverture retenues ne sont pas respectées à l'issue des tests fonctionnels, la conception des tests structurels complémentaires nécessite donc d'analyser le programme pour essayer de déterminer les jeux de données qui permettraient d'atteindre cette couverture.
Ainsi, lorsqu'on se rend compte que les tests fonctionnels n'étaient pas complets on définit les jeux de données nécessaires et on effectue de nouveaux tests fonctionnels complémentaires.

V – Le Cahier des tests unitaires

Ce document doit présenter l'ensemble des trois catégories de tests effectués ainsi que les éventuelles anomalies détectées et corrigées.

Plan du Cahier des tests unitaires

1 – Introduction
- Nature du document,
- acteurs mis en jeux,
- objectifs du document, ...

2 - La revue de code
- Description de la démarche de revue de code
- Découpage du code en fonctions et répartition des lecteurs,
- Rapports sur le code faits par les lecteurs.

3 - Les tests fonctionnels

On présente dans ce chapitre, fonction par fonction, l'ensemble des spécifications et résultats des réalisations des tests fonctionnels. On expose aussi les éventuels problèmes rencontrés et corrections réalisées sur le code.

4 - Les tests structurels

Dans ce chapitre, on présente les graphes de contrôle construits par fonction, la vérification des règles de couverture, et les tests fonctionnels complémentaires éventuellement effectués (ainsi que les problèmes et corrections).

5 - Validation de la conception

Ce chapitre doit dresser un bilan et permettre de certifier que les modules réalisés correspondent bien à la conception qu'on en avait fait dans la phase de conception du logiciel.

6 - Glossaire
7 - Références
8 - Annexes
9 - Index

Chapitre 9 : Intégration et validation du logiciel

I – Objectifs

L'objectif de cette partie va être de valider les travaux de la phase de spécification du logiciel. Autant la phase des tests unitaires visait à valider la décomposition hiérarchique (axe vertical de l'arbre ordonné des modules), autant cette phase conduit à valider les relations entre modules de même niveau (axe horizontal). Dans cette phase on s'attache plus à vérifier que le logiciel a une exécution cohérente, qu'à vérifier qu'il s'exécute sans planter.

II – La démarche

On intègre dans un premier temps les modules de bas niveau pour constituer des entités dont le niveau d'abstraction est de plus en plus important. Lorsque l'on arrive à des modules tels qu'ils peuvent être considérés comme peu dépendants du niveau d'abstraction supérieur alors, suivant la procédure établie lors de la réalisation de tests fonctionnels on établit des tests de validation de l'unité en lui donnant en entrée des objets cohérents avec le domaine d'application dans lequel le logiciel opérera. C'est la grande différence par rapport à la phase des tests unitaires : les tests réalisés permettent de vérifier la cohérence des manipulations faites sur les données par le logiciel.

EXEMPLE : Dans un logiciel de gestion de la comptabilité d'une entreprise, un module permet de gérer les bilans annuels. C'est un module de forte abstraction pour lequel on générera des données cohérentes avec l'activité de l'entreprise.

Ainsi par intégrations et validations successives on arrive à valider complètement le logiciel par rapport à ses spécifications. Si le système à concevoir était constitué de plusieurs logiciels interagissant entre eux alors cette étape va permettre de valider les logiciels dans leur globalité. Cette validation est interne à l'équipe de travail.

III – Le Cahier d'Intégration et Validation du logiciel

C'est ce document qui va permettre de certifier que le logiciel réalisé correspond bien à ce que l'équipe de travail en attendait au travers des spécifications du logiciel.

Plan du Cahier d'Intégration et Validation du Logiciel

1 – Introduction
- Nature du document,
- acteurs mis en jeux,
- objectifs du document, ...

2 - Les tests de validation interne

Ce chapitre contient l'ensemble des modules testés, les jeux de données cohérents avec le domaine d'application (éventuellement leur provenance et la justification de leur pertinence), les anomalies constatées.

3 - Synthèse des problèmes rencontrés

Ce chapitre doit contenir une synthèse des problèmes rencontrés, leur source (problèmes de spécification, de conception ou de codage ?), les coûts et retards induits ainsi que les solutions à apporter pour les résoudre.

4 - Validation de la spécification du logiciel

Ce chapitre doit dresser un bilan et permettre de certifier que le logiciel réalisé correspond bien à la spécification qu'on en avait fait dans la phase de spécification du logiciel.

5 - Glossaire

6 - Références

7 - Annexes

8 - Index

IV – Rédaction des manuels d'utilisation

C'est à partir de cette phase qu'on peut commencer à rédiger l'ensemble des manuels d'utilisation. Ils seront terminés lors de la phase d'intégration et de validation du système. [Strohmeier-1996] pour un plan de rédaction d'un manuel utilisateur.

Chapitre 10 : Intégration et validation du système

I – Objectifs

> Cette phase va dans la continuité de la précédente. Il s'agit de replacer le logiciel sur son infrastructure matérielle pour reconstituer le système et pouvoir ainsi le valider. De même, le client peut avoir spécifié des *tests de recette* (tests qui lui permettent de constater que le système correspond à ce qu'il en attendait) que l'on va réaliser dans cette phase. Une fois que le système est validé il peut aussi être mis à l'épreuve durant une certaine période chez le client (une version bêta lui est alors fournie). L'objectif clairement formulé de cette phase est de vérifier que le système réalisé correspond bien à ce que le client en attendait, autrement dit à valider la phase de spécification du système.

La diversité des cas de figure étant importante il n'est pas possible de délivrer une démarche de validation externe. Si des tests de recette (spécifiés lors de la phase de spécification du système) sont à effectuer on peut réutiliser la même procédure que pour les tests fonctionnels (en intégrant les desiderata du client) pour construire les tests de recette. Si une version bêta est livrée au client, il faut alors assurer un suivi chez celui-ci...
En tout état de cause, dès que le système est validé par le client, on considère qu'il est apte à entrer en situation réelle de fonctionnement.

II – Le Cahier d'Intégration et Validation du système

Plan du Cahier d'Intégration et Validation du Système

1 – Introduction
- Nature du document,
- acteurs mis en jeux,
- objectifs du document, ...

2 - Les tests de validation externe

Ce chapitre contient l'ensemble des tests de recette réalisés, des erreurs éventuellement rencontrées ainsi que des solutions proposées. Si une version bêta du système a été confiée au client, une synthèse du suivi et des remarques éventuelles est à fournir.

3 - Validation de la spécification système

Ce chapitre doit dresser un bilan et permettre de certifier que le logiciel réalisé correspond bien à ce qu'en attendait le client.

4 - Glossaire
5 - Références
6 - Annexes
7 - Index

Chapitre 11 : La maintenance

I – Présentation

La phase de maintenance est supposée réalisée par des programmeurs spécialisés, utilisant des méthodes et outils spécifiques. La maintenance de programme constitue une activité de conception, comme celui qui a permis l'écriture initiale du programme. Elle suit donc les étapes du cycle de vie du logiciel.

On distingue quatre types de maintenance [Strohmeier-1996] :

- maintenance corrective (correction de « bugs »),
- maintenance perfective (amélioration des performances ou du design),
- maintenance adaptative (adaptation à un autre environnement d'une version existante du système),
- maintenance évolutive (faire évoluer le système).

Une perception classique de la maintenance est que l'on fasse essentiellement de la maintenance corrective. D'après le SWEBOCK, ce n'est pas tout à fait le cas puisque 80% des activités de maintenance vont essentiellement porter sur de la maintenance évolutive ou adaptative.

Faire de la maintenance c'est, avant tout, être capable de comprendre le code écrit. Un code complexe, mal écrit, mal documenté sera donc long à comprendre et sa maintenance n'en sera que plus couteuse : on estime justement que la moitié de l'effort de maintenance est dédiée à la compréhension du logiciel à modifier ! (SWEBOCK 2022).

II – Maintenance des anciens programmes

La principale difficulté dans la maintenance des programmes anciens résulte du fait que ceux-ci ne sont souvent ni bien structurés, ni modulaires, ni documentés.
Pour maintenir un tel programme, la simple réécriture de la documentation est en général illusoire car l'expérience montre que le souci d'écrire alors une documentation claire et structurée conduit à une description inexacte si le code lui-même n'est pas structuré. La seule méthode consiste à restructurer le code. Il ne s'agit pas de réeffectuer, à partir de rien, toute l'analyse des besoins à satisfaire, mais d'exprimer, sous une forme différente, des idées, des fonctions et des solutions déjà existantes qui souvent satisfaisaient déjà ces besoins.

III – Représentation et évolution des programmes

Pour modifier un système il est nécessaire de le comprendre, c'est-à-dire de repasser par les mêmes étapes d'abstraction que celles qui se sont avérées nécessaires pour le concevoir. Il ne faut jamais séparer les différentes représentations du système. Ainsi, la documentation élaborée au cours du processus initial de développement va devenir obsolète dès que le système va évoluer et la seule représentation qu'il en restera sera le code. C'est pour éviter ce

genre de problèmes que l'on considère qu'un système est indissociable de sa documentation et que l'évolution de l'un entraîne l'évolution de l'autre. L'évolution de la documentation consiste à créer un nouveau document, le Cahier des évolutions, dans lequel on trouvera non seulement une description du système modifié mais aussi des transformations que cela a impliqué.

De même, modifier un système nécessite de remettre en oeuvre un processus de développement partiel. Pour cette raison, on ne peut pas dissocier non plus le système des outils utilisés pour le concevoir (programmes de test, ...).

IV – La gestion des versions

La modularité dans la conception logicielle induit l'existence de problèmes de mise à jour des modules, du suivi de ces mises à jour et de l'intégration dans le système. Comment faire évoluer un module ? Comme garantir la compatibilité ascendante d'un module ayant évolué vis-à-vis du système duquel il a été tiré ? Comment savoir si un module de version X et compatible avec un système de version Y ?

La gestion des versions est un problème épineux qui nécessite une grande rigueur. Le contrôle de version est un processus qui permet de gérer les numéros de version des modules et systèmes logiciels. De récents travaux [Plaice et al-1993] proposent de définir une grammaire, au sens théorie des langages, qui permettrait non seulement de définir avec précision les caractéristiques d'une version, mais aussi de déterminer à partir d'un numéro de version d'un système quelles sont les versions des modules le constituant à considérer.

Chapitre 12 : Notion et évaluation de la qualité

I - Evaluation de la qualité du logiciel [Menthonnex-1996a][Pham Van-1986]

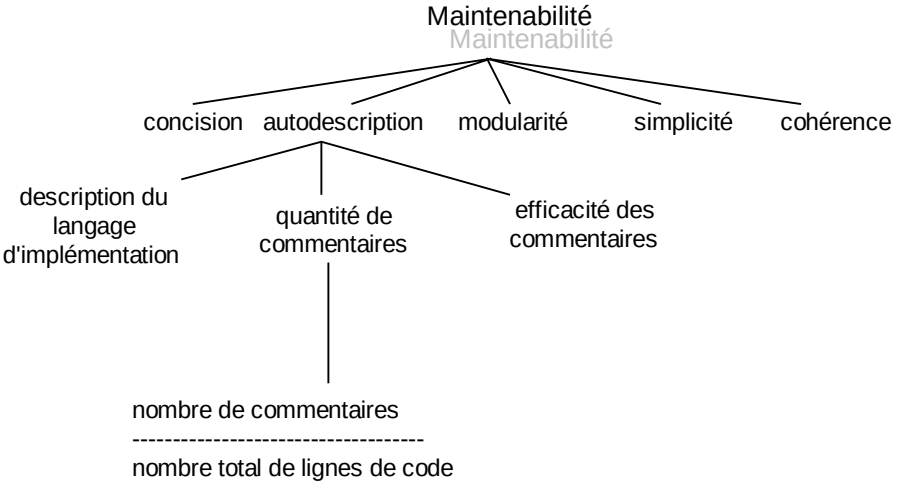
1°) Le modèle de qualimétrie de la norme ISO/IEC 9126

Un modèle de qualimétrie est une représentation de la réalité permettant de qualifier la qualité d'un système (Est-il bon, moyen, mauvais, ... ?). Face à l'émergence du nombre de modèles, le besoin d'établir une norme dans ce domaine est devenu impératif. La norme ISO/IEC 9126 propose donc un modèle normalisé de qualimétrie.

Les six caractéristiques de qualité et leurs définitions sont les suivantes :

- capacité fonctionnelle : ensemble d'attributs portant sur l'existence d'un ensemble de fonctions et leurs propriétés données. Les fonctions sont celles qui satisfont aux besoins exprimés ou implicites,
- fiabilité : ensemble d'attributs portant sur l'aptitude du logiciel à maintenir son niveau de service dans des conditions précises et pendant une durée déterminée,
- facilité d'utilisation : ensemble d'attributs portant sur l'effort nécessaire pour l'utilisation et sur l'évaluation individuelle de cette utilisation par un ensemble défini ou implicite d'utilisateurs,
- rendement : ensemble d'attributs portant sur le rapport existant entre le niveau de service d'un logiciel et la quantité de ressources utilisées, dans des conditions déterminées,
- maintenabilité : ensemble d'attributs portant sur l'effort nécessaire pour faire des modifications données,
- portabilité : ensemble d'attributs portant sur l'aptitude du logiciel à être transféré d'un environnement à l'autre.

Ces six facteurs ne sont pas détaillés ici. Pour obtenir la liste complète des attributs se reporter à la norme ISO/IEC 9126. Néanmoins, nous allons illustrer le facteur de maintenabilité [Menthonnex-1996a] :



- Fig. 30 : évaluation du facteur de maintenabilité -

2°) Des métriques pour la quantification de la qualité

Ces métriques, de nature empirique, ont pour objectif de permettre aux développeurs de mesurer la qualité du logiciel produit en termes de nombre d'erreurs, de fiabilité et de complexité. On distingue communément deux types d'analyse : l'analyse statique (métrique de Halstead et métrique de McCabe par exemple) et l'analyse dynamique (métrique de Mohanty, ...). La première constitue une aide à l'inspection du code et à la conception de tests unitaires. La seconde permet de garder l'historique de l'exécution du programme durant le test, à savoir : chemins testés ? modules testés ? ce que l'on a testé l'était-il facilement ? A-t-on réussi à tester un chemin difficilement testable ?

NOTE : L'analyse brute des résultats retournés par ces métriques est très difficile et elle ne peut être menée qu'à l'aide d'une base de connaissance des valeurs possibles qui sert alors de référentiel. Ainsi, par la pratique, **l'équipe de travail va se constituer une base de données des valeurs obtenues, par indice et par métrique de façon à pouvoir appréhender plus facilement les tests des projets futurs en se positionnant par rapport aux projets passés.**

Nous aborderons dans ce paragraphe quelques métriques de base.

a)Mesure de la complexité de l'architecture d'un logiciel [Menthonnex-1996a] [Mohanty-1979]

La métrique de Mohanty repose sur la théorie des graphes et permet d'évaluer la complexité de l'architecture d'un logiciel, c'est-à-dire le degré de lisibilité de celle-ci.

L'architecture d'un logiciel peut être représentée sous forme d'un graphe d'appel. On rappelle qu'un graphe G=<X,U> est composé d'un ensemble de sommets X et d'un ensemble d'arcs U.

Le graphe d'appel d'un logiciel est construit de la manière suivante :

- tout composant (fonction, module) est représenté par un sommet $x_i \in X$,
- tout lien d'appel entre deux fonctions (sommets x_i et x_j) est représenté par un arc $u_k=(x_i ; x_j) \in U$.

Une fois que le graphe est modélisé, on utilise différentes mesures :

- le **nombre de chemins**. Un chemin étant le parcours d'un sommet racine vers un sommet terminal,
- la **complexité hiérarchique** (nombre moyen de composants par niveau). C'est le nombre de composants sur le nombre de niveaux du graphe,
- la **complexité structurelle** (nombre moyen d'appels par composants). C'est le nombre d'appels (les arcs) sur le nombre de composants (les sommets),

Les mesures suivantes sont des probabilités (donc comprises entre 0 et 1) :

- l'**accessibilité d'un composant** :

$$A(M_i) = \sum_j A(M_j) * q_{ji} * p_j$$

avec M_j les composants appelant M_i, q_{ji} la probabilité de passage de M_j à M_i et p_j la probabilité que le composant M_j s'exécute correctement. En réalité, il est souvent difficile d'estimer q et p. On choisit alors de fixer p à 1 (tous les M_j s'exécutent correctement) et q à l'équiprobabilité (tous les composants étant liés à M_i ont autant de chance de se servir de M_i).

ILLUSTRATION : Si seuls M_1, M_2 et M_3 sont utilisés par M_4 alors les q_{4i}=1/3.

Cette hypothèse simplificatrice conduit à

$$A(M_i) = \sum_j \frac{A(M_j)}{N_j}$$

avec N_j le nombre de composants appelés par M_j. L'accessibilité du point d'entrée vaut 1. Plus $A(M_i)$ est faible et moins le composant sera jugé accessible.

- la **testabilité d'un chemin**.

$$T(P) = \left(\sum_{M_i \in P} \frac{1}{A(M_i)} \right)^{-1}$$

où P est le chemin considéré. Cette mesure permet d'évaluer si un chemin peut être facilement testable (est-il souvent parcouru ? ...). Pour un chemin P, plus la valeur T(P) est importante et plus ce chemin sera facilement testable.

- la **testabilité du système**.

$$T(S) = \frac{1}{N} \left(\sum_{i=1}^{N} \frac{1}{T(P_i)} \right)^{-1}$$

où N est le nombre de chemins et P_i un chemin. Plus la valeur T(S) est importante est plus le système sera testable.

- l'**entropie du système**.

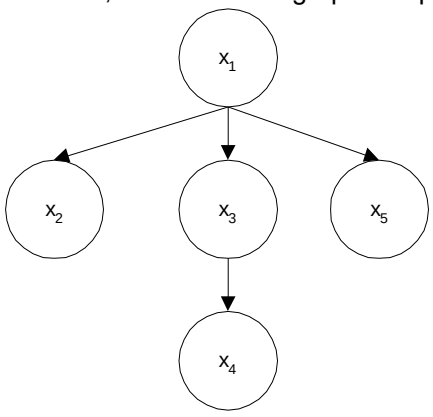
$$H(Ga) = \frac{1}{X} \sum_{i=1}^{N} (X_i \log_2 (\frac{X}{X_i}))$$

avec N le nombre de chemins, X_i le nombre de composants du chemin P_i et X la somme des X_i.

EXEMPLE : Prenons l'exemple d'un logiciel effectuant la résolution d'équations du second degré (trinômes) de la forme $y=ax^2+bx+c$. On a les fonctions suivantes :

→Résoudre_trinôme (sommet x_1),
→Saisir_trinôme (sommet x_2),
→Calculer_discriminant (sommet x_3),
→Calculer_solutions (sommet x_4),
→Afficher_solutions (sommet x_5).

Après étude des liens entre ces fonctions, on construit le graphe d'appel suivant :



- Fig. 31 : graphe d'appel d'un système de résolution de trinômes -

Ici le noeud racine est x_1 (Résoudre_trinôme).

Les chemins sont au nombre de N=3 : $P_1=(x_1 ; x_2)$ / $P_2=(x_1 ; x_3 ; x_4)$ / $P_3=(x_1 ; x_5)$.

On a 3 niveaux dans le graphe et 5 composants. La **complexité hiérarchique** est donc de 5/3=1,667. Cela signifie qu'en moyenne on a 1,667 composants par niveau. La **complexité structurelle** est de 4/5=0,2.

Pour calculer l'**accessibilité d'un composant** on remarque d'abord que :

$A(x_1)$=1 car la fonction Résoudre_trinôme est appelée dès l'exécution du logiciel (donc toujours accessible <=> probabilité d'accessibilité =1). On se place de plus dans le cas de l'hypothèse

simplificatrice émise précédemment, ce qui conduit à :
A(x_2)=A(x_1)/N_1=1/3=0,333 / **A(x_3)**=A(x_1)/N_1=1/3=0,333
A(x_4)=A(x_3)/N_3=0,333/1=0,333 / **A(x_5)**=A(x_1)/N_1=1/3=0,333
Autrement dît, tous les composants ont la même chance d'être atteints.
La **testabilité des chemins** est la suivante :
T(P_1) = 1/(1/1 + 1/0,333) = ¼ = 0,25
T(P_2) = 1/(1/1 + 1/0,333 + 1/0,333) = 1/7 = 0,143
T(P_3) = 1/(1/1 + 1/0,333) = ¼ = 0,25
La **testabilité du système** est donc de :
T(S) = 1/3*1/(1/4 + 1/7 + 1/4) = 1/3 * 14/9 = 14/27 = 0,518
Enfin, l'**entropie du système** est de :
H(Ga)=1/7*(2*1,807+3*1,222+2*1,807)=1,556

On remarque que plus la testabilité des chemins va être forte (donc facilement testables aux vues de l'architecture) et plus la testabilité du système va être forte. On en conclue donc que plus la testabilité d'un chemin est grande et plus celui-ci pourra être validé lors des tests. De même, plus la testabilité du système sera grande et plus le système sera validable.
Cette métrique est utilisée en phase de test unitaire des composants.

b)Mesure de la complexité des composants [Menthonnex-1996a][McCabe-1976]

Pour mesurer la complexité d'un composant on peut utiliser la métrique de Mc Cabe. Comme pour la structure d'un programme (cf. métrique précédente), un composant peut être représenté sous la forme d'un graphe de contrôle (cette notion a déjà été abordée dans le chapitre 3, paragraphe VII-3, page 70). On rappelle qu'un graphe de contrôle est construit à partir des structures élémentaires de contrôle du langage de programmation.

De façon analogue à la métrique de Mohanty, différentes mesures permettent d'évaluer la complexité d'un composant. La métrique de McCabe repose sur la théorie des graphes pour définir la notion de **nombre cyclomatique** v(G) dans un graphe de contrôle G. Ce nombre est défini par :

$$v(G) = e - n + 2$$

où e est le nombre d'arcs et n le nombre de sommets. Ce nombre exprime le nombre de cycles élémentaires indépendants du graphe, c'est-à-dire des chemins qui passent une fois par un arc et une fois par un sommet (chemins du début à la fin). On considère alors, par expérience, qu'un composant logiciel a une complexité facilement gérable si le nombre cyclomatique est inférieur ou égal à 15.
Trois autres mesures sont complémentaires au nombre cyclomatique :

- le **nombre maximum de degré** :

  $D = Max (D_i)$

  avec D_i degré d'un sommet i : D_i = nombre d'arcs entrants et sortants.

  Si $D \geq 4$ alors on est certain qu'il existe une structure **selon...cas** dans le composant.
- le **nombre maximum d'imbrications** I :

  il se définit comme étant le nombre maximum de structures de contrôles imbriquées en cascade. Une norme généralement admise consiste à considérer que ce nombre ne doit pas dépasser 4.
- la **densité de contrôle** C :

  C = v(G) / nombre de noeuds

  elle fournit un indicateur brut de branchement ramené au nombre de blocs séquentiels.

EXEMPLE : Pour le pseudo-code suivant :

```
Si x Alors
   Tant que y
        b
   Fin Tantque
  Sinon
     z
Fin Si
a
```

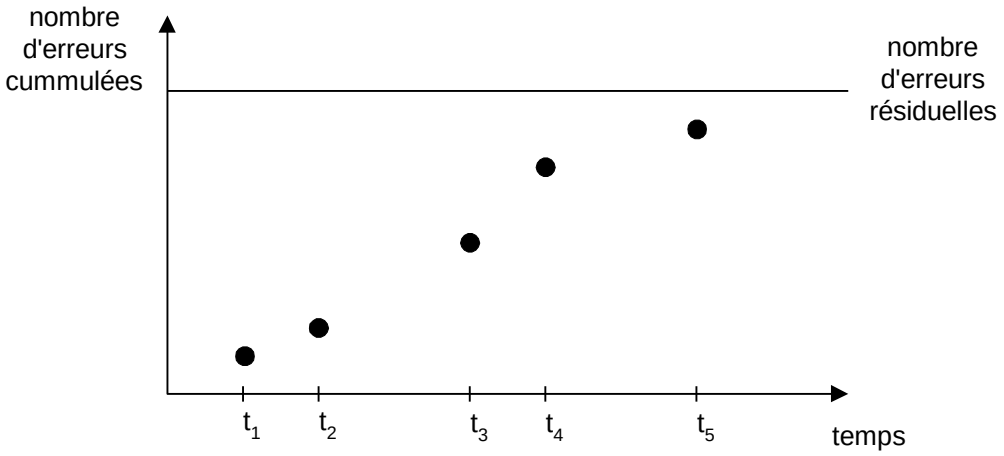
On a v(G)=6-5+2=3 chemins.

Ainsi, la connaissance de ces mesures permet d'appréhender la structure du composant logiciel et donc apporte une information quant à sa qualité. Cette métrique est utilisée dans la phase de test unitaire des composants.

c)Mesure de la qualité d'un module en terme d'erreurs [Menthonnex-1996a] [Halstead-1977]

A partir du nombre d'opérateurs et d'opérandes utilisés dans le code source, cette métrique permet de connaître l'encombrement physique d'un module à partir du nombre de lignes de code source. Le principal avantage de cette métrique est de permettre le suivi du nombre d'erreurs présentes dans un programme.

ILLUSTRATION : Supposons qu'un programme P soit constitué de 5 modules M_1 à M_5. Prenons le cas du module M_1. Lors de son codage, des « bugs » y figurent et seront corrigés au fils de la phase de tests et ultérieurement. La métrique de Halstead propose, entre autre, un indicateur d'erreurs résiduelles ce qui permet de connaître l'état de correction d'un module.



- Fig. 32: état de correction du module M_1 -

Pour pouvoir appliquer la métrique de Halstead sur un code source quatre données sont nécessaires :

- n_1 : nombre d'opérateurs types (du code),
- n_2 : nombre de type d'opérandes (du code),
- N_1 : nombre d'occurrences des opérateurs dans le code,
- N_2 : nombre d'occurrences des opérandes dans le code.

Les principales mesures proposées sont les suivantes :

- taille du vocabulaire : $n=n_1 + n_2$,
- longueur du programme : $N=N_1 + N_2$,
- volume du programme : $V=N*\log_2(n)$,

Cette mesure exprime le nombre minimal d'éléments binaires nécessaires à un programme de longueur N ayant n symboles distincts. Ce volume devrait être compris entre 20 et 1000. Au-delà de 1000 il faut certainement songer à découper la fonction en sous-fonctions.

- niveau du programme : $L=2*n_2 / (n_1*N_2)$,
  Plus cette mesure est basse et plus le niveau du programme est bas, i.e plus le risque d'erreurs est important.
- niveau du langage : $NL = L^2 * V$,
- difficulté du programme : $D=1/L$,
- effort mental pour le concevoir : $E = V / L$,
  Cette mesure reflète l'effort de conception qu'il est nécessaire pour concevoir le code correctement.
- Temps pour le concevoir : $T= E/18$,
  Cette mesure reflète une estimation du temps en secondes pour concevoir ce code. La valeur 18 a été obtenue par Halstead suite à des expérimentations.
- nombre d'erreurs résiduelles : $Ner=E^{2/3}/3000$.
  Cette mesure permet d'estimer le nombre d'erreurs présent en moyenne dans un tel code.

Cette métrique pose un problème important : supposons qu'après plusieurs modifications le module M_1 que nous considérions précédemment soit totalement refondu, voir éclaté en plusieurs modules. C'est un facteur que la métrique ne prend pas en compte dans l'évaluation du nombre d'erreurs résiduelles et qu'il semble difficile de contourner. Autrement dit, cette métrique est peu sensible à l'organisation du code. Elle constitue néanmoins un bon indicateur sur sa complexité.

EXEMPLE : Soit un code, écrit en langage C, qui permette de rechercher un élément dans une liste :

```
/*On suppose qu'en entrée du module sont fournies :
        la liste d'entier L, sa longueur n et k l'entier à
        rechercher.*/
int i, trouve;
trouve=0; /*l'élément k n'est pas trouvé*/
i=0;
while ((i<n)&&(trouve==0))
{
  if (L[i]==k) trouve=1; /* L'élément k est trouvé*/
  i=i+1;
}
if (trouve==1) printf(« Le nombre a été trouvé\n »);
    else printf(« Le nombre n'a pas été trouvé\n »);
```

Les données relatives à ce code sont :

- $n_1 = 10$ (=, while, <, &&, if, +, printf, else, ==, []),
- $n_2 = 4$ (constante, variable, expression, paramètre),
- $N_1 = 17$,
- $N_2 = 26$.

Les mesures sont alors :

- taille du vocabulaire : n=**14**,
- longueur du programme : N=**43**,
- volume du programme : $V=43*\log_2(14)$=**163,71**,
- niveau du programme : $L=2*4 / (10*26)$=**0,0307**,
- niveau du langage : $NL = 0.0307^2 * 163,71$=**0,54**,
- difficulté du programme : $D=1/0.0307$=**32,57**,
- effort mental pour le concevoir : E = 163,71 / 0.0307 = **5332,573**,
- temps pour le concevoir : T= 5332,573/18=**296,254 secondes (environ 5mn)**,

- nombre d'erreurs résiduelles : Ner=5332,573$^{2/3}$/3000=**0,101 bug**.

Etant donné le peu d'information qu'on possède sur des projets passés et même sur les résultats pour d'autres modules du même projet il est difficile de donner une interprétation aux chiffres ci-dessus. La seule chose qu'on peut en déduire est que le nombre d'erreurs résiduelles est proche de 0 i.e sur un code aussi simple et petit il ne doit pas y avoir d'erreurs.

Prenons maintenant un autre exemple, celui d'une fonction nommée *calculer_schrage()* et qui fait 54 lignes de code effectives (hors commentaires). On a :

- n_1 = 20,
- n_2 = 4,
- N_1 = 145,
- N_2 = 258.
- taille du vocabulaire : n=**24**,
- longueur du programme : N=**403**,
- volume du programme : V=403*$\log_2$(24)=**1847,755**,
- niveau du programme : L=2*4 / (20*258)=**0,00155**,
- niveau du langage : NL = 0.00155^2 * 1847,755=**0,00444**,
- difficulté du programme : D=1/0.00155=**645,161**,
- effort mental pour le concevoir : E = 1847,755/ 0.00155 = **1192100**,
- temps pour le concevoir : T= 1192100/18=**66227,77 secondes (environ 18h de travail… surestimation de la réalité plus proche de 7h-8h)**,
- nombre d'erreurs résiduelles : Ner=1192100$^{2/3}$/3000=**3,747 bugs (pas loin de la vérité)**.

3°) Interprétation des résultats et rapports d'analyse

Le processus d'analyse de la qualité d'un logiciel s'effectue en plusieurs étapes :

1 - On étudie tout d'abord l'architecture de l'application à l'aide de la métrique de Mohanty. En particulier, on peut voir si le graphe possède plusieurs racines (il faut alors s'interroger sur leur pertinence). Il est également possible de constater si des composants se trouvent isolés (il faut certainement les supprimer). Les points importants seront les composants fortement sollicités, les récursivités, les sauts de niveaux (signes de problème de conception).

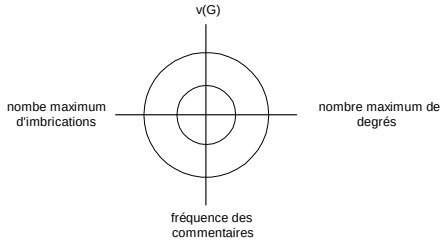
2 - L'étude des graphes de contrôle, métrique de McCabe, des composants logiciels est la seconde étape de l'analyse. Comme un projet peut s'avérer gigantesque il peut être fastidieux et trop coûteux de visualiser les composants un par un. C'est pour cela que la plupart des outils logiciels de qualimétrie permettent de restreindre le champ d'investigation en filtrant les composants qui dépassent les normes imposées pour le projet. On n'oubliera pas de rajouter les composants que l'on a identifiés à l'étape précédente. Mis à part les dépassements des valeurs imposées aux mesures effectuées sur le graphe de contrôle, on pourra visualiser les parties de code non atteignables, les problèmes de non structuration (v(G)>3), la présence de similitudes graphiques (qui peut signifier que l'on a effectué un « copier-coller » auquel cas une factorisation peut permettre de réduire le nombre cyclomatique du graphe mais ceci aux dépends du graphe d'appel).

3 - Appliquer la métrique de Halstead et commencer un suivi des erreurs corrigées.

On peut citer comme logiciel statistique de mesure de la qualité des logiciels, le logiciel Logiscope de Visilog.

Le rapport d'analyse doit rendre compte du suivi de la qualité effectué. Il comporte donc le graphe d'appel, les graphes de contrôle des composants logiciels qui posent problème ainsi que le résultat de toutes les mesures effectuées (y compris celles de la métrique de Halstead).

Un moyen de présenter ces résultats est le graphe de Kivial où le cercle inscrit représente la borne minimale des mesures et le cercle extérieur, la borne supérieure. Les valeurs trouvées pour le projet sont donc reportées sur ce graphe et doivent se situer, pour un logiciel de qualité acceptable, entre les bornes inférieures et supérieures.



- Fig. 33 - un diagramme de Kiviat -

Annexe 1 : Les mantras du Génie Logiciel

I – Principes fondamentaux

1°) Enoncé des principes.

Voici cinq principes fondamentaux du Génie Logiciel :

PSR - Principe de séparation des responsabilités :

L'architecture d'un logiciel doit être séparée en différentes parties chacune gérant un aspect précis de la problématique. Chaque partie regroupera un ensemble de modules.

PA - Principe d'abstraction :

Chaque unité modulaire (fichier, module, classe, …) doit correspondre à une abstraction préexistante et doit pouvoir être définie de façon abstraite, indépendamment de sa réalisation. On lui associe également un rôle et une responsabilité qui permettent de définir son périmètre fonctionnel.

PE - Principe d'encapsulation :

La description logique d'une unité modulaire (fichier, module, classe, …) doit permettre de cacher systématiquement toute sa complexité interne tout en fournissant les informations suffisantes pour la comprendre et l'utiliser.

PFC - Principe de faible couplage :

A chaque niveau d'analyse, les rapports entre unités modulaires (modules, classes, fonctions, …) qu'il est nécessaire de considérer doivent être simples.

PU - Principe d'uniformité :

Tous les éléments d'un logiciel devront suivre les mêmes règles : (i) convention de nommage, (ii) élément de syntaxe, (iii) écriture des interfaces qui définissent l'accès aux services, (iv) type et ordre des paramètres, …

Ces principes ont des conséquences à de nombreux moments du cycle de vie d'un logiciel. Ce sont comme des mantras à garder en tête et chercher à toujours respecter.

2°) Ilustration

Reprenons le cas d'application déjà introduit dans le cours, en phase de spécification du système.

ILLUSTRATION : Votre Client est Polytech Tours et vous sollicite pour développer un logiciel qui permette de gérer les emplois du temps d'une école d'ingénieurs : on a donc une spécialité (informatique, ….), des cours (CMs, TDs, TPs), des étudiants répartis par groupes éventuellement, des salles et des professeurs. L'objectif est que le système permette aux scolarités de créer les emplois du temps de leur spécialité et aux étudiants de consulter leur emploi du temps. Evidemment, ce logiciel va s'appuyer sur le système d'information existant au sein de l'école, c'est-à-dire sur le logiciel Apogée pour accéder aux maquettes des spécialités, à la liste des enseignants et à la liste des salles.

En phase de spécification du logiciel, le principe **PSR** va impliquer que l'on sépare l'architecture du logiciel en plusieurs parties :

- Gestion des salles,
- Gestion des enseignants,
- Gestion des maquettes,
- Gestion des emplois-du-temps.

Chaque partie regroupant un ou plusieurs modules/classes.

Le principe **PA** implique d'abstraire des entités du domaine d'application pour créer les modules/classes (du moins, la plupart pas toutes ne pourront être identifiées comme cela). Si on s'intéresse à tout ce qui concerne la gestion des enseignants, on va donc interroger la *scolarité* pour savoir s'il faut informatiser des « fiches enseignants » et quelles informations y rattacher : il faudra donc sans doute créer un module/classe **Enseignant**. Le *rôle* de ce module/classe sera de représenter un enseignant amené à faire un cours. La *responsabilité* du module/classe est par conséquent de permettre au reste du programme de faire tous les traitements de base sur un enseignant (modifier/ajouter des informations sur un enseignant). Cela définit donc son *périmètre fonctionnel*.

Le principe **PE** a plusieurs conséquences :

- Au niveau d'un module/classe comme **Enseignant**, il faudra introduire des méthodes pour permettre au code utilisateur de modifier ou accéder aux informations (attributs) de l'enseignant. Il faudra également interdire au code utilisateur d'accéder directement à ces informations. (**encapsulation des données + pratique FEV**)
- Au niveau d'un module/classe comme **Enseignant**, il faudra également que le code utilisateur ait accès à toutes les méthodes dont il aura besoin et que l'utilisation de ces méthodes soient le plus indépendant possible de leur implémentation. Une méthode=Un service offert (**encapsulation des traitements**).
- Au niveau du programme, il faut identifier quelles sont les Entrées/Sorties que celui-ci va réaliser. Notamment, il va y avoir des interactions avec le Système d'Information pour récupérer ou sauvegarder des données. Il faudra prévoir des modules/classes spécifiques pour ces E/S : par exemple, au moins un module **Enseignant_SI** pour gérer l'écriture/lecture d'un enseignant dans le SI et pour générer un objet **Enseignant** manipulé par le reste du programme. Seules les modules/classes d'E/S doivent gérer les interactions avec l'environnement, le reste du programme doit en être déconnecté (**encapsulation des Entrées/Sorties**).

Le principe **PFC** va avoir des conséquences sur les liens entre les modules/classes. Notamment, dans les phases de conception du logiciel et codage. Supposons que nous ayons un module/classe **Emploi-du-Temps** qui, pour une spécialité et une année d'études données, contiennent l'emploi-du-temps associé pour toutes les semaines de l'année. Au sein de ce module/classe vous avez la méthode suivante :

```
AjouterEDT(Salle, Enseignant, Matière, Volume_horaire, Type)
```

Avec `Type` valant : 1 si c'est un CM, 2 si c'est un TD, et 3 si c'est un TP.
La définition de cette méthode n'est pas compatible avec **PFC** (ni à la pratique **COM**) car l'appel à la méthode est trop complexe : le dernier argument, `Type`, est un argument de type *option* car il va conditionner de façon optionnelle le déroulement de la méthode. Il vaut mieux dupliquer la méthode pour simplifier son utilisation :

```
AjouterCM_EDT(Salle, Enseignant, Matière, Volume_horaire)
AjouterTD_EDT(Salle, Enseignant, Matière, Volume_horaire)
AjouterTP_EDT(Salle, Enseignant, Matière, Volume_horaire)
```

II – Bonnes pratiques

1°) Enoncé des principes.

S'appuyant sur les principes fondamentaux, une démarche de Génie Logiciel prône le développement d'un logiciel qui poursuive plusieurs bonnes pratiques :

REU – Réutilisabilité du code produit :

Pour le projet en cours, **il faut chercher à identifier les modules/classes d'intérêt général : ils seront réutilisés s'ils existent**. S'ils n'existent pas, leurs spécifications seront élargies par rapport aux stricts besoins du projet en cours de façon à faciliter leur réutilisation ultérieure.

FEV – Favoriser l'évolution du code produit :

Le choix d'une architecture modulaire nécessite une réflexion préalable sur les changements de spécification auxquels aura à faire face le programme produit (évolutions prévisibles).

Les évolutions prévisibles devront être encapsulées dans un minimum de modules/classes. L'interface de ces modules/classes permettra de cacher les parties du code qui peuvent évoluer au fil des versions (encapsulation des modifications).

COM – Minimiser la complexité :

Pour faciliter la compréhension et l'écriture du code, on préfère toujours écrire des codes simples et facilement lisibles. On évitera les structures de code trop complexe. **La factorisation de code est un élément important permettant de réduire sa complexité.** Notez que les patrons de conception sont une approche très factorisée de la conception d'un programme en orienté objets.

VER – Vérifiabilité du code produit :

L'écriture du code doit faciliter la vérifiabilité de sa correction : il faut donc suivre les standards de codage en vigueur, organiser son développement pour y inclure les tests unitaires autant que possible. Il faut également éviter l'utilisation de structures du langage trop compliquées à comprendre et utiliser.

PRM – Protection modulaire :

Tout module/classe est responsable des erreurs survenant à l'occasion de son déroulement. Une spécification d'exception, figurant dans son interface, définit les anomalies dont il assure l'identification ainsi que la méthode utilisée pour en informer son client.

2°) Illustration

Continuons avec l'exemple précédent.

Que va impliquer la règle de bonne pratique **REU** ? Supposez que dans votre logiciel vous ayez besoin de gérer des listes d'enseignants. Vous avez donc le module/classe **Enseignant** et le module/classe **Liste_Enseignants**. Bien ! Maintenant, vous avez analysé le problème et vous vous rendez compte que vous aurez toujours besoin d'ajouter en dernier dans la liste et

vous retirer toujours le premier élément ajouté (votre liste est donc gérée en mode FIFO). La conséquence est que vous allez écrire, par exemple, la méthode Ajouter comme suit :

`AjouterEnseignant(Enseignant)`

Après, 5mn de réflexion, vous en arrivez à la conclusion que **Liste_Enseignants** est un module/classe d'intérêt général car il pourrait être réutilisé dans d'autres logiciels liés à l'éducation national ou l'enseignement supérieur. Donc, pour favoriser sa réutilisabilité, vous allez plutôt élargir les spécifications du module/classe ce qui va avoir pour conséquence que la méthode Ajouter va devenir :

`AjouterEnseignant(Enseignant,Position)`

où le paramètre `Position` va indiquer à quelle position dans la liste vous ajoutez l'objet enseignant. Qui peut le plus, peut le moins !

Après encore 5mn de réflexion, vous réalisez que dans un autre projet informatique vous aviez déjà développé un module/classe **Liste** qui permettait de gérer des listes dont le type des éléments est quelconque. Donc ce module/classe peut gérer des listes d'enseignants, de salles, … bingo ! Vous allez donc plutôt réutiliser directement ce module/classe.

Cet exemple illustre les deux facettes de **REU**.

Maintenant illustrons la règle **COM**, et notamment sa dimension *factorisation de code*. Voici un exemple de code, sans lien avec notre cas d'utilisation précédent.

```
3   int main()
4   {
5       unsigned int uiBoucle, uiBoucle2, uiFactoriel;
6       double dArrangement, dCombinaison;
7
8       for (uiBoucle = 1; uiBoucle < 10; uiBoucle++)
9       {
10          if (uiBoucle % 2 == 0)
11          { // L'indice est un nombre pair nous allons donc calculer un arrangement de uiBoucle parmi 10
12              // On calcule 10!
13              uiFactoriel = 1;
14              for (uiBoucle2 = 1; uiBoucle2 <= 10; uiBoucle2++) uiFactoriel *= uiBoucle2;
15              dArrangement = uiFactoriel;
16              // On calcule (10-uiBoucle)! et on divise l'arrangement par cette valeur
17              uiFactoriel = 1;
18              for (uiBoucle2 = 1; uiBoucle2 <= (10-uiBoucle); uiBoucle2++) uiFactoriel *= uiBoucle2;
19              dArrangement /= (double)uiFactoriel;
20          }
21          else
22          { // L'indice est un nombre impair nous allons donc calculer une combinaison de uiBoucle parmi 10
23              // On calcule 10!
24              uiFactoriel = 1;
25              for (uiBoucle2 = 1; uiBoucle2 <= 10; uiBoucle2++) uiFactoriel *= uiBoucle2;
26              dCombinaison = uiFactoriel;
27              // On calcule (10-uiBoucle)! et on divise la combinaison par cette valeur
28              uiFactoriel = 1;
29              for (uiBoucle2 = 1; uiBoucle2 <= (10 - uiBoucle); uiBoucle2++) uiFactoriel *= uiBoucle2;
30              dCombinaison /= (double)uiFactoriel;
31              // On calcule (uiBoucle)! et on divise la combinaison par cette valeur
32              uiFactoriel = 1;
33              for (uiBoucle2 = 1; uiBoucle2 <= uiBoucle; uiBoucle2++) uiFactoriel *= uiBoucle2;
34              dCombinaison /= (double)uiFactoriel;
35          }
36      }
37  }
```

Ce code est trop complexe et difficilement lisible alors qu'il fait quelque chose de très simple. Déjà on remarque que les lignes 12-14 et 23-25 font la même chose et que ce traitement est indépendant de la valeur de uiBoucle. On va faire une première factorisation….

```c
int main()
{
    unsigned int uiBoucle, uiBoucle2, uiFactoriel;
    double dArrangement, dCombinaison;

    for (uiBoucle = 1; uiBoucle < 10; uiBoucle++)
    {
        if (uiBoucle % 2 == 0)
        { // L'indice est un nombre pair nous allons donc calculer un arrangement de uiBoucle parmi 10
            // On calcule 10!
            uiFactoriel = 1;
            for (uiBoucle2 = 1; uiBoucle2 <= 10; uiBoucle2++) uiFactoriel *= uiBoucle2;
            dArrangement = uiFactoriel;
            // On calcule (10-uiBoucle)! et on divise l'arrangement par cette valeur
            uiFactoriel = 1;
            for (uiBoucle2 = 1; uiBoucle2 <= (10-uiBoucle); uiBoucle2++) uiFactoriel *= uiBoucle2;
            dArrangement /= (double)uiFactoriel;
        }
        else
        { // L'indice est un nombre impair nous allons donc calculer une combinaison de uiBoucle parmi 10
            // On calcule 10!
            uiFactoriel = 1;
            for (uiBoucle2 = 1; uiBoucle2 <= 10; uiBoucle2++) uiFactoriel *= uiBoucle2;
            dCombinaison = uiFactoriel;
            // On calcule (10-uiBoucle)! et on divise la combinaison par cette valeur
            uiFactoriel = 1;
            for (uiBoucle2 = 1; uiBoucle2 <= (10 - uiBoucle); uiBoucle2++) uiFactoriel *= uiBoucle2;
            dCombinaison /= (double)uiFactoriel;
            // On calcule (uiBoucle)! et on divise la combinaison par cette valeur
            uiFactoriel = 1;
            for (uiBoucle2 = 1; uiBoucle2 <= uiBoucle; uiBoucle2++) uiFactoriel *= uiBoucle2;
            dCombinaison /= (double)uiFactoriel;
        }
    }
}
```

```c
int main()
{
    unsigned int uiBoucle, uiBoucle2, uiFactoriel, uiFactoriel10;
    double dArrangement, dCombinaison;

    // On calcule 10!
    uiFactoriel10 = 1;
    for (uiBoucle2 = 1; uiBoucle2 <= 10; uiBoucle2++) uiFactoriel10 *= uiBoucle2;

    for (uiBoucle = 1; uiBoucle < 10; uiBoucle++)
    {
        if (uiBoucle % 2 == 0)
        { // L'indice est un nombre pair nous allons donc calculer un arrangement de uiBoucle parmi 10
            dArrangement = uiFactoriel10;
            // On calcule (10-uiBoucle)! et on divise l'arrangement par cette valeur
            uiFactoriel = 1;
            for (uiBoucle2 = 1; uiBoucle2 <= (10-uiBoucle); uiBoucle2++) uiFactoriel *= uiBoucle2;
            dArrangement /= (double)uiFactoriel;
        }
        else
        { // L'indice est un nombre impair nous allons donc calculer une combinaison de uiBoucle parmi 10
            dCombinaison = uiFactoriel10;
            // On calcule (10-uiBoucle)! et on divise la combinaison par cette valeur
            uiFactoriel = 1;
            for (uiBoucle2 = 1; uiBoucle2 <= (10 - uiBoucle); uiBoucle2++) uiFactoriel *= uiBoucle2;
            dCombinaison /= (double)uiFactoriel;
            // On calcule (uiBoucle)! et on divise la combinaison par cette valeur
            uiFactoriel = 1;
            for (uiBoucle2 = 1; uiBoucle2 <= uiBoucle; uiBoucle2++) uiFactoriel *= uiBoucle2;
            dCombinaison /= (double)uiFactoriel;
        }
    }
}
```

En regardant le code, on constate qu'il y a plein de portions identiques : ce sont toutes celles qui calculent un factoriel. On peut factoriser tout ça en introduisant une fonction `Factoriel()`.

```
3   double Factoriel(unsigned int uiRacine)
4   {
5       unsigned int uiBoucle;
6       double dFactoriel = 1;
7
8       for (uiBoucle = 0; uiBoucle <= uiRacine; uiBoucle++)
9           dFactoriel *= (double)uiBoucle;
10
11      return dFactoriel;
12  }
13
14  int main()
15  {
16      unsigned int uiBoucle, uiBoucle2, uiFactoriel, uiFactoriel10;
17      double dArrangement, dCombinaison;
18
19      // On calcule 10!
20      uiFactoriel10 = Factoriel(10);
21
22      for (uiBoucle = 1; uiBoucle < 10; uiBoucle++)
23      {
24          if (uiBoucle % 2 == 0)
25          { // L'indice est un nombre pair nous allons donc calculer un arrangement de uiBoucle parmi 10
26              dArrangement = uiFactoriel10 / Factoriel(10-uiBoucle);
27          }
28          else
29          { // L'indice est un nombre impair nous allons donc calculer une combinaison de uiBoucle parmi 10
30              dCombinaison = uiFactoriel10 / (Factoriel(10 - uiBoucle)*Factoriel(uiBoucle));
31          }
32      }
33  }
34
```

En on aurait même pu encore simplifier en remarquant que pour calculer une combinaison il faut d'abord calculer un arrangement puis diviser par (`uiBoucle !`). Bien sûr, on pourrait même utiliser la fonction du langage C qui calcule un factoriel, plutôt que redéfinir notre propre fonction. En tout cas, le code est beaucoup plus simple maintenant !

Bibliographie

Généralités :

[Boehm-1998] B. W. Boehm : A spiral model of software development and enhancement, IEEE Computer, vol 21, n°5, pp. 61-72, 1998

[Calvez-1990] J.P. Calvez : Spécification et conception des systèmes : une méthodologie, Ed. Masson, 610 p., 1990

[Gaudel et al-1996] M.-C. Gaudel, B. Marre, F. Schlienger, G. Bernot : Précis de génie logiciel, Ed. Masson, 128 p., 1996

[Jensen et al-1979] R.W. Jensen, C.C. Tonies : Software Engineering, Ed. Prentice-Hall International Editions, 1979

[Ramamoorthy et al-1987] C.V. Ramamoorthy, A. Prakash, V. Garg, T. Yamaura, A. Bhide : Issues in the development of large, distributed, and reliable software, Advances in Computers, vol 26, pp 393-443, 1987

[Wilson-1986] B. Wilson : Systems : Concepts, methodologies and applications, Ed. John Wiley &Sons, New York, 1986

[Strohmeier-1996] A. Strohmeier : Cycle de vie du logiciel, dans A. Strohmeier, D. Buchs (Eds.), Génie logiciel : principes, méthodes et techniques, Ed. Presses polytechniques et universitaires romandes, pp. 1-28, 1996

Conduite de Projets et assurance qualité:

[Cours CP] Cours de Conduite de Projets, Ecole d'Ingénieurs en Informatique pour l'Industrie

[Halstead-1977] M. H. Halstead : Elements of Software Science, Ed. Elsevier, 1977

[McCabe-1976] T.J. McCabe : A complexity Measure, IEEE Transactions on Software Engineering, vol SE-2, n°4, pp. 308-320, Décembre 1976

[Pham Van-1986] N. Pham Van : Rôle des métriques de complexite d'un programme source dans l'assurance qualité logiciel, 5° colloque international de fiabilité et de maintenabilité, Biarritz (France), pp. 75-80, 1986.

[Menthonnex-1996a] J. Menthonnex : Principes du management de projet dans les développements informatiques, dans A. Strohmeier, D. Buchs (Eds.), Génie logiciel : principes, méthodes et techniques, Ed. Presses polytechniques et universitaires romandes, pp. 29-40, 1996

[Menthonnex-1996b] J. Menthonnex : Concepts et principes de la qualité totale. Leur application aux développements logiciels, dans A. Strohmeier, D. Buchs (Eds.), Génie logiciel : principes, méthodes et techniques, Ed. Presses polytechniques et universitaires romandes, pp. 291-311, 1996

[Mohanty-1979] S. N. Mohanty : Models and measruements for quality assessment of software, Computing surveys, vol 11, n°3, 1979

[Raffy-1996] J.-L. Raffy : La qualité du logiciel : son évaluation à l'aide de la métrologie, dans A. Strohmeier, D. Buchs (Eds.), Génie logiciel : principes, méthodes et techniques, Ed. Presses polytechniques et universitaires romandes, pp. 242-258, 1996

Les méthodes du Génie Logiciel :

[Alford et al-1982] M.W. Alford, J.P. Ansart, G. Hommel, L. Lamport, B. Liskov, G.P. Mullery, F.B. Schneider : Distributed systèms. Methods and Tools for specification, Lectures notes in Computer science, Ed. Springer-Verlag, 1982

[Alford-1985] M. W. Alford : SREM at the age of Eight ; The distributed Computing Design System, Computer, pp. 36-46, Avril 1985

[Bianciotto et al-?] Bianciotto, Boye : L'informatique en automatisation industrielle, Ed. Delagrave
[Bolognesi et al-1988] O. Bolognesi, Ed. Briksma : Introduction to the ISO Specification Language LOTOS, ISDN, vol 14, n°1, pp. 25-29 (1988)
[Booch-1983] G. Booch : SoftwareEngineering with ADA, Ed. Benjamin/Cummings, Menlo Park, CA., 1983
[Booch-1986] G. Booch : Object-Oriented Development, IEEE Transactions on Software Engineering, vol SE-12, n°2, pp.211-221, 1986
[Brams-1983] G.W. Brams : réseaux de Pétri : théorie et pratique, Ed. Masson, 2 tomes, tome 1 : théorie et analyse, tome 2 : modélisation et applications, 1983
[Buhr-1984] R.J.A. Buhr : System Design with ADA, Ed. Prentice-Hall, 1984
[Buhr-1989] R.J.A. Buhr : System Design with Machinie Charts : A CAD approach with ADA examples, Ed. Prentice-Hall, 1989
[Calvez-1990] J.P. Calvez : Spécification et conception des systèmes : une méthodologie, Ed. Masson, 610 p., 1990
[Cours AutoSeq] Cours d'Automatisme Séquentiel, Ecole d'Ingénieurs en Informatique pour l'Industrie
[Cours BD] Cours de Base de Données, Ecole d'Ingénieurs en Informatique pour l'Industrie.
[Cours Merise] Cours de Merise, Ecole d'Ingénieurs en Informatique pour l'Industrie
[Cours OMT] Cours d'OMT, Ecole d'Ingénieurs en Informatique pour l'Industrie
[Cours RdPs] Cours de Réseaux de Pétri, Ecole d'Ingénieurs en Informatique pour l'Industrie
[Cours SADT] Cours de SADT, Ecole d'Ingénieurs en Informatique pour l'Industrie
[Cox-1986] B.S. Cox : object oriented programming : an evolutionnary approach, Ed. Addsion-Wesley, 1986
[De Marco-1978] T. De Marco : Structured Analysis and System Specification, Ed. Prentice Hall, 1978
[Gooma-1984] H. Gomaa : A software design méthode for the real-time sytems, Communications of the ACM, vol 27, n°9, 1984
[Gooma-1989a] H. Gomaa : A software design methode for distributed real-time applications, Journal of systems and software, n°9, pp.81-94, 1989
[Gooma-1989b] H. Gomaa : Structuring criteria for real-time systems design, Proceedings of the 11th International Confrence on Software Engineering, pp.290-301, 1989
[Hatley et al-1987] D.J. Hatley, I.A. Pirbhai : Strategies for Real-time System Specification, Ed. Dorset House Publishing, New-York, 1987
[Heitz-1987] M. Heitz : HOOD, une méthode de conception hiérarchisée orientée objets pour le développement des gros logiciels techniques et temps réel, Bigre n°57, Journées ADA, France, pp. 42-61, 1987
[Jackson-1983] M. Jackson : System development, C.A.R. HOARE series, Ed. Prentice-Hall, 1983
[Jaulent-1992] P. Jaulent : Génie Logiciel : les méthodes, Ed. Armand Colin, 1992
[Lissandre-1990] M. Lissandre : Maîtriser SADT, Ed. Armand Colin (Paris), 1990
[Marco-1978] T. de Marco : Structured Analysis and System Specification, Ed. Prentice Hall, 1978
[Mourlin-1996] F. Mourlin : Présentation du langage LOTOS, dans A. Strohmeier, D. Buchs (Eds.), Génie logiciel : principes, méthodes et techniques, Ed. Presses polytechniques et universitaires romandes, pp. 159-206, 1996

[Racloz-1996] P. Racloz : Introduction aux réseaux de Pétri, dans A. Strohmeier, D. Buchs (Eds.), Génie logiciel : principes, méthodes et techniques, Ed. Presses polytechniques et universitaires romandes, pp. 207-240, 1996

[Rumbaugh et al-1995] J. Rumbaugh, Blaha M., Premerlani W., Eddy F., Lorensen W. : O.M.T., modélisation et conception orientées objet, Ed. Masson et Ed. Prentice-Hall, 1995

[Ward et al-1985] P.T. Ward, S.J. Mellor : Structured Development for real-time systems, vol 1 : Introduction and Tools, vol 2 : Essential modeling Techniques, vol 3, Implementation modeling techniques, Ed. Prentice hall, 1985

Conception des interfaces Hommes/Machines :

[Cours Ergo] Cours d'hygiène et sécurité du travail/Ergonomie des postes de travail, Ecole d'Ingénieurs en Informatique pour l'Industrie

[Cours SIAD] Cours de Système Interactif d'Aide à la Décision, Ecole d'Ingénieurs en Informatique pour l'Industrie

[Grize-1996] F. Grize : Conception et réalisation d'interfaces-utilisateurs graphiques, dans A. Strohmeier, D. Buchs (Eds.), Génie logiciel : principes, méthodes et techniques, Ed. Presses polytechniques et universitaires romandes, pp. 83-103, 1996

[Ukelson et al-1996] J.P. Ukelson, D. Gould, J. Boies : User navigation in computer application, IEEE Transactions on Software Engineering, vol 19, n °3, pp. 297-306, 1993

Réalisation de programmes modulaires :

[AFNOR-1986] Système graphique de base (GKS) description fonctionnelle, Norme Française NF-EN 27942, Afnor (1986)

[Cours AS] Cours d'algorithmique et structure des données, Ecole d'Ingénieurs en Informatique pour l'Industrie

[Dijkstra-1976] E.W. Dijkstra : A discipline of Programming, Ed. Prentice-Hall, Englewood Cliffs (1976)

[Knight et al-1986] J.-C. Knight, N.G. Leveson : An experimental evaluation of assumption of independance in multiversion programming, IEEE Transaction on Software Engineering, SE-12 (1986)

[Meyer-1988] B. Meyer : Object-oriented software construction, Ed. Prentice-Hall, International series in computer science, Hemel Hempstead (1988)

[Parnas-1972] D.L. Parnas : On the criteria to be used in decomposing systems into modules, Communications of the ACM, vol 15, n°12 (1972)

[Parnas-1984] D.L. Parnas : Software engineering principles, Informatic, vol 22, n°4 (1984)

[Pierra-1991] G. Pierra : Les bases de la programmation et du génie logiciel, Ed. Dunod informatique, 653 p., 1991

Tests et validation de programmes :

[Lleres et al-1989] J. LLeres, M. Monavon, D. Mahe : Le test des logiciels régulation moteur pour une certification DO-178A : principe et expérience, BIGRE+GLOBULE, vol 60 (1989)

[Ostrand et al-1988] T.H. Ostrand, M.J. Balcer : The category-partition method for specifying and generaling functional tests, Communications of the ACM, vol 31, n°6 (1988)

[Pierra-1991] G. Pierra : Les bases de la programmation et du génie logiciel, Ed. Dunod informatique, 653 p., 1991

Maintenance de programmes :

[Plaice et al-1993] J. Plaice, W.W. Wadge : A new approach to version contrôle, IEEE Transactions on software engineering, vol 19, n°3 (Mars 1993).